TURING 图灵程序设计丛书

第2版

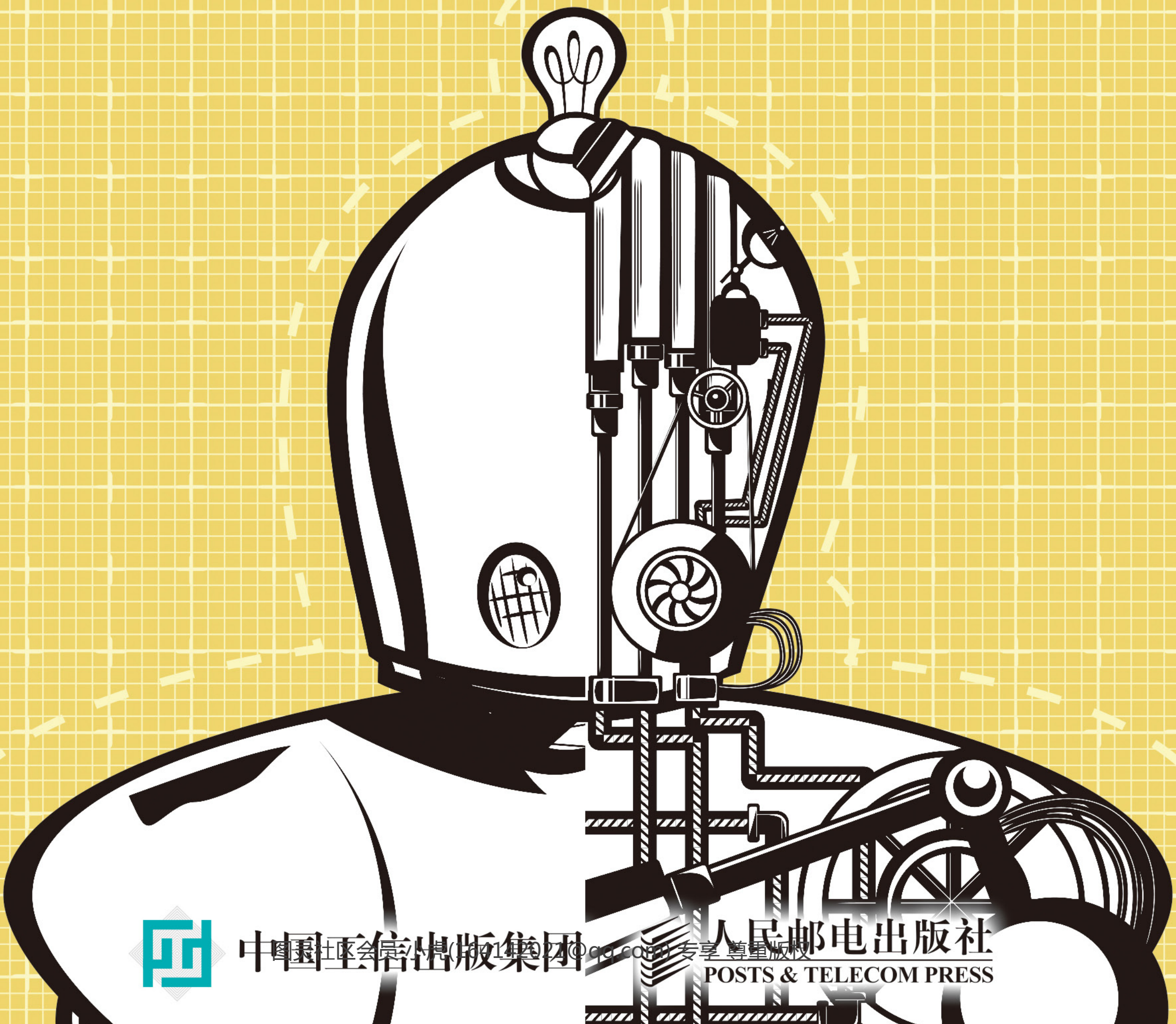
How Linux Works

2nd Edition

What Every Superuser Should Know

精通Linux

【美】Brian Ward 著　姜南 袁志鹏 译



中国工信出版集团　人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

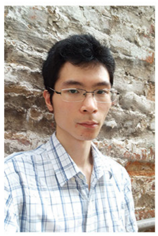
Brian Ward

毕业于芝加哥大学，获计算机科学博士学位。1993年开始接触Linux，研究并教授Linux二十余年。除了这本深受读者喜爱的图书，Brian还著有*Linux Kernel–HOWTO*（No Starch Press）、*The Linux Problem Solver*（No Starch Press）等畅销著作。现居旧金山，身兼计算机高级顾问与高级讲师等数职。



姜南

老码农一名，编程之外喜欢骑行、读书和大海，目前生活在悉尼。从事过国内外金融、通信、移动互联网、数字出版等领域的IT系统开发，擅长的技术有Java、C#、HTML5、JavaScript、iOS、Android。长期在独立技术博客——酷壳（coolshell.cn）发表翻译文章，目前为图灵的签约译者。



袁志鹏

2012年7月信息管理专业本科毕业，至今有半年银行业AS400经验，两年半航运业Java、Oracle、UNIX开发经验，开发企业内部业务系统。

图灵程序设计丛书

How Linux Works

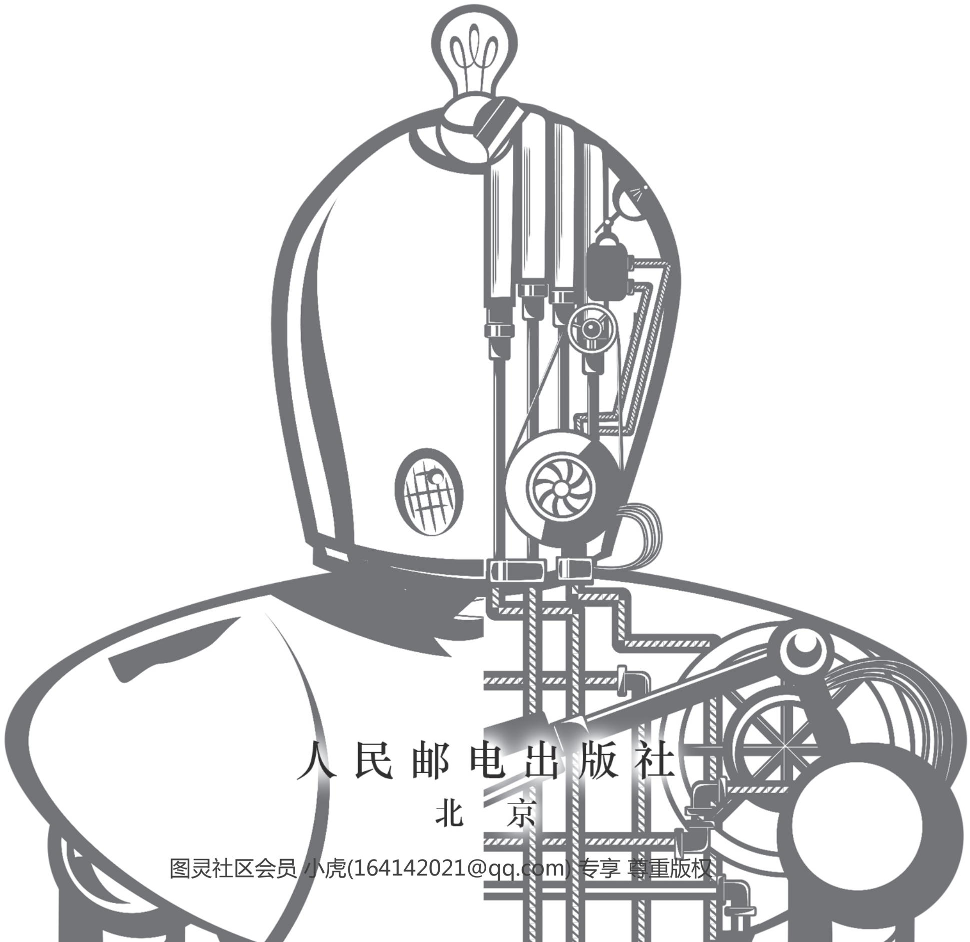
2nd Edition

What Every Superuser Should Know

精通Linux

第2版

【美】Brian Ward 著　姜南 袁志鹏 译



人民邮电出版社
北　京

图书在版编目（C I P）数据

精通Linux : 第2版 / (美) 沃德 (Ward,B.) 著 ;
姜南, 袁志鹏译. -- 北京 : 人民邮电出版社, 2015.7
（图灵程序设计丛书）
ISBN 978-7-115-39492-7

Ⅰ. ①精… Ⅱ. ①沃… ②姜… ③袁… Ⅲ. ①
Linux操作系统 Ⅳ. ①TP316.89

中国版本图书馆CIP数据核字(2015)第122015号

内 容 提 要

本书讲解了 Linux 操作系统的工作机制以及运行 Linux 系统所需的常用工具和命令。根据系统启动的大体顺序，本书更深入地介绍从设备管理到网络配置的各个部分，最后演示了系统各部分的运行方式，并介绍了一些基本技巧和开发人员常用的工具。

本书适合对 Linux 感兴趣的初学者阅读。

◆ 著　　　　[美] Brian Ward
译　　　　姜　南　袁志鹏
责任编辑　岳新欣
执行编辑　张　曼
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：19
字数：449千字　　　　2015年 7 月第 1 版
印数：1－4 000册　　　2015年 7 月北京第 1 次印刷

著作权合同登记号　图字：01-2014-8584号

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

前　　言

之所以写这本书，是因为我觉得你应该对你使用的计算机有所了解。你应该可以让软件去做你想让它去做的事（当然要在它的能力范围之内）。要做到这一点，关键是必须理解软件能做什么，以及是怎么做的。这些正是本书要介绍的内容。这样你就不必对着计算机抓狂了。

如果你要学习这方面的知识，Linux是一个很好的平台，因为它是一个透明的系统。特别是大多数系统配置都存放在文本文件中，让人一目了然。难点在于了解每个组件分别负责什么，以及它们如何协同工作。

读者对象

我们学习Linux的原因可能各不相同。对于IT从业者（如系统运维人员）来说，他们需要了解本书中的几乎所有内容。对于Linux软件架构师和开发人员来说，他们同样需要了解这些内容，以便发挥操作系统的最大功效。对于只需考虑个人所用Linux系统的研究人员和学生来说，本书能够让他们理解为什么系统是那个样子的。

还有一些堪称多面手的读者，出于兴趣、谋利或其他原因而摆弄计算机，喜欢探究事情的根源，喜欢尝试不同的可能性。或许你就是其中一位。

阅读要求

虽然开发人员热爱Linux，不过并非开发人员才能阅读本书，只要你有一些基础的计算机知识即可。也就是说，你需要知道如何操作GUI（特别是能看懂各种Linux发行版的安装和配置界面），需要知道什么是文件什么是目录（文件夹）。此外，还要有心理准备随时查看系统文档或者上网搜索一些相关文章。当然，我前面提过，最重要的是你对电脑的热情。

阅读方法

要对任何技术建立起系统的认识都不是件容易的事。而说到软件系统的工作原理，就更复杂了。有时候面对大量的技术细节，读者会难以抓住重点（因为人类大脑无法同时处理太多新概念），但是如果解释得不够透彻，又会让读者一知半解，不利于后面的学习。

本书每一章都会先介绍最重要最基本的知识，以便让读者能够继续深入。为了突出重点，有

些地方简化了很多内容。随着一章内容的展开，更多细节才会在最后几节出现。这些内容需要你马上就掌握吗？通常不用，我经常也会这么提醒你。如果你觉得正在阅读的内容有些枯燥难懂，可以随时跳到下一章或者稍事休息。

动手操作

你最好准备一台可以用来实际操作的Linux计算机。你可以使用虚拟机，比如我就使用VirtualBox来测试本书中的很多实例。你需要拥有超级用户（root）权限，不过多数情况下你需要以普通用户身份登录系统。我们将主要通过终端窗口或者远程会话来运行命令行。如果你之前毫无经验也无大碍，书中第2章会让你尽快上手。

书中的命令通常是像下面这样：

```
$ ls /
[输出结果]
```

你只需输入第一行粗体的文本，非粗体的文本是系统的输出结果。$是普通用户提示符。如果你是超级用户的话则是#。（详见第2章。）

本书结构

本书分为三个部分。第一部分整体介绍Linux系统以及运行Linux系统所需的常用工具和命令。随后我们会根据系统启动的大体顺序，更深入地介绍从设备管理到网络配置的各个部分。最后我们会演示系统各部分的运行方式，并介绍一些基本技巧和开发人员常用的工具。

除第2章以外，开始的几章均主要讲解Linux内核，然后逐步涉及用户空间。（如果你现在对我所说的一头雾水也没关系，我们将在第1章中介绍这些概念。）

本书的内容尽量保证对各个版本的Linux系统均适用。但要涵盖各个系统之间的差异也实在是项繁琐的工作，所以我尽量考虑两个主要的Linux版本：Debian（包括Ubuntu）和RHEL/Fedora/CentOS。本书主要针对的是桌面和服务器系统。嵌入式系统（如Android和OpenWRT）也多有涉及，但各系统之间的差异还需要你自己去探索。

第 2 版的新内容

本书的第1版侧重于从用户的角度来介绍Linux系统，旨在帮助读者了解系统各部分的工作原理。彼时Linux上的软件安装和配置还不是那么容易。

有幸的是，随着各种新版本的出现，这些问题已然不复存在，所以我剔除了一些较为陈旧和不太相关的内容（比如打印），以便能够更加深入地介绍Linux内核。你可能没有意识到你将会多么频繁地和内核打交道。

当然，上一版中的很多内容随着时间推移也发生了较大变化，我花了大量的精力梳理和更新

了它们，特别是在Linux的启动和设备管理方面。我对很多内容也进行了重新组织，以满足当下读者的阅读兴趣与需要。

本书没有发生变化的是它的厚度。我希望读者能够尽快上手，因此会解释一些不太容易理解的细节，但我又不想让这本书变得你拿都拿不动。只要掌握了本书介绍的知识，自己再去深入探索就不是一件难事了。

我还删掉了第1版中一些关于历史背景的介绍，目的是突出重点。如果你对Linux和Unix的历史感兴趣，可以参考Peter H. Salus所著*The Daemon, the Gnu, and the Penguin*（Reed Media Services，2008），这本书详细介绍了我们使用的各种软件的历史沿革。

关于术语

关于操作系统中某些组件应该叫什么，一直都存在争论。甚至“Linux”是否应该叫作“GNU/Linux”也存在争论，因为其中使用了GNU项目的成果。本书中我们尽量使用通用术语，不使用拗口、生硬的词汇。

致　　谢

感谢以下对本书的第 1 版提供过帮助的人：James Duncan、Douglas N. Arnold、Bill Fenner、Ken Hornstein、Scott Dickson、Dan Ehrlich、Felix Lee、Scott Schwartz、Gregory P. Smith、Dan Sully、Karol Jurado 以及 Gina Steele。在第 2 版的写作中，我要特别感谢 Jordi Gutiérrez Hermoso 卓越的技术审阅工作，他为本书提供了极有价值的建议和勘误。还要感谢 Dominique Poulain 和 Donald Karon，他们在本书写作期间就给出了非常好的反馈意见，还要感谢 Hsinju Hsieh 在我写作这本书期间对我的宽容。

最后，我还要感谢本书策划编辑 Bill Pollock、项目编辑 Laurel Chun，以及 No Starch Press 出版社的 Serena Yang、Alison Law 和其他人为本书面世所做的一如既往的卓越工作。

第 1 版书评

“非常棒的书。在近 350 页的内容中涵盖了 Linux 的所有基础知识。”

——EWEEK

“对于那些想要学习 Linux，同时对操作系统内部工作原理又不太熟悉的读者，本书绝对值得推荐。”

——O’REILLYNET

“介绍 Linux 基础知识最好的书之一，同时也适合 Linux 高级用户阅读，五星。”

——OPENSOURCE-BOOK-REVIEWS.COM

“本书的成功源于它对内容的良好组织和对技术细节的深入探讨。”

——KICKSTART NEWS

“本书对 Linux 的介绍可谓标新立异。它朴实无华，注重对命令行的介绍，并且深入到系统内部，而非仅仅停留在图形用户界面。”

——TECHBOOKREPORT.COM

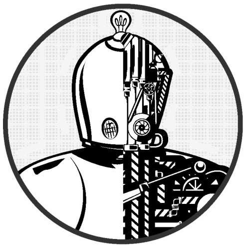
“本书很好地介绍了 Linux 系统的工作原理。”

——HOSTING RESOLVE

目　　录

第 1 章

概　　述



乍看起来，Linux这样的现代操作系统非常复杂，内部有多得令人眼花缭乱的各种组件在同步运行和相互通信。比如：Web服务器可以连接到数据库服务器，还有可能用到很多其他程序也在使用的公共组件。那么，整个系统究竟是怎样运作的呢？

理解操作系统工作原理最好的方法是*抽象思维*，换句话说，你可以暂时忽略大部分细节。就像坐车一样，通常你不会去在意车内固定发动机的装配螺栓，也不会关心你走的路是谁修筑的。如果你是一个乘客的话，你可能只关心车要做的事情（比如车要把你带到哪）以及车的一些基本操作（比如如何打开车门、怎样系好安全带）。

但如果你在开车的话，就需要了解更多的细节，比如如何控制油门、怎样换挡，还有如何处理意外情况。

如果我们觉得开车这个事情太复杂，就可以运用“抽象思维”来帮助理解。首先你可以将“一辆汽车在路上行驶”抽象为三个部分：汽车、道路和驾驶操作。这样有助于将复杂的问题分解开来。如果道路颠簸，你不会去埋怨车辆本身和你的驾驶技术。相反，你可能会问为什么这条路这么烂，或者如果这是条新修的路的话，那么筑路工人的活干得可真够差劲的。

软件开发人员运用抽象思维来开发操作系统和应用程序。在计算机软件领域有许多术语来描述抽象的子系统，如*子系统*、*模块*和*包*等。本书中我们使用*组件*这个相对简单的词。在软件开发过程中，开发人员通常不用太关心他们需要使用的组件的内部结构，他们只关心能使用哪些组件，以及怎么个用法。

本章概述了Linux操作系统涉及的主要组件。虽然每一个组件都包含纷繁复杂的技术细节，但我们将暂时忽略这些细节，而专注于这些组件在系统中发挥的功能。

1.1 Linux 操作系统中的抽象级别和层次

在组织得当的前提下，通过抽象将系统分解为组件有助于我们了解其工作机制。我们将组件划分为*层次*或*级别*。组件的层次（或级别）代表它在用户和硬件系统之间所处的位置。Web浏览器、游戏等应用处于最高层，底层则是计算机硬件系统，如内存。操作系统处于这两层之间。

Linux操作系统主要分为三层。如图1-1所示，最底层是硬件系统，包括内存和中央处理器（用于计算和从内存中读写数据），此外硬盘和网络接口也是硬件系统的一部分。

硬件系统之上是*内核*，它是操作系统的核心。内核是运行在内存中的软件，它向中央处理器发送指令。内核管理硬件系统，是硬件系统和应用程序之间进行通信的接口。

*进程*是指计算机中运行的所有程序，由内核统一管理，它们组成了最顶层，称为*用户空间*。（另一个更确切的术语是用户进程，无论它们是否直接和用户交互。例如，所有的Web服务器都是以用户进程的形式运行的。）

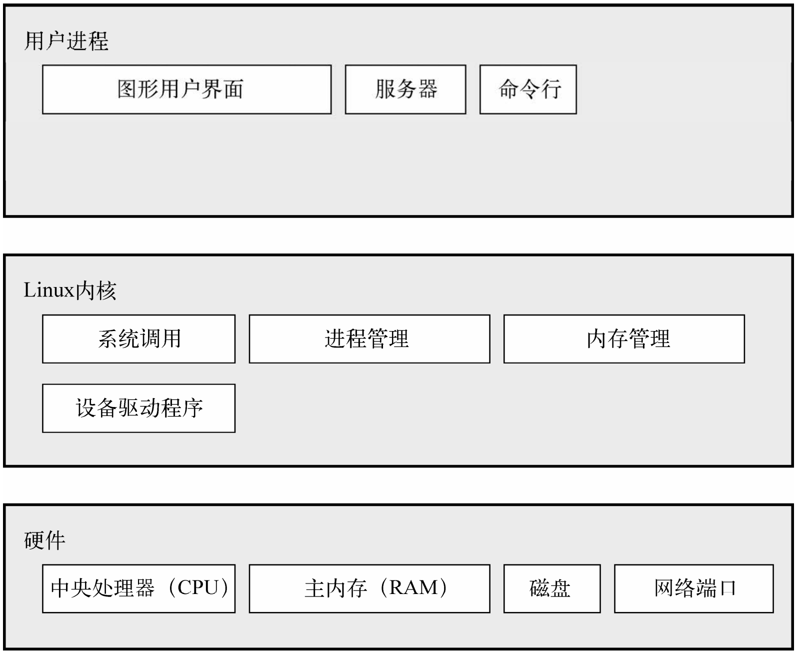


图1-1　Linux系统的基本组成

内核和用户进程之间最主要的区别是：内核在*内核模式*（kernel mode）中运行，而用户进程则在*用户模式*（user mode）中运行。在内核模式中运行的代码可以不受限地访问中央处理器和内存，这种模式功能强大，但也非常危险，因为内核进程可以轻而易举地使整个系统崩溃。那些只有内核可以访问的空间我们称为*内核空间*（kernel space）。

相对于内核模式，用户模式对内存和中央处理器的访问有一定程度的限制，可访问的内存空

间通常很小，对CPU的操作也很安全。用户空间指的是那些用户进程能够访问的内存空间。如果一个用户进程出错并崩溃的话，其导致的后果也相对有限，并且能够被内核清理掉。例如，如果你的Web浏览器崩溃了，不会影响到你正在运行的其他程序。

理论上来说，一个用户进程出问题并不会对整个系统造成严重的影响。当然这取决于我们如何定义“严重的影响”，并且还取决于该进程拥有的权限。因为不同的进程拥有的权限可能不同，一些进程能够执行一些别的进程无权执行的操作。举个例子，如果拥有足够的权限，用户进程可以将硬盘上的数据全部清除。也许你会觉得这样太危险，但好在操作系统提供了一些相关的安全措施，而且大多数用户进程并没有这个权限。

1.2　硬件系统：理解主内存

主内存（main memory）或许是所有硬件系统中最为重要的部分。基本上来讲，主内存存储0和1这样的数据。我们将每个0和1称为一个比特（或位，bit）。内核和进程就在主内存中运行，它们就是一系列比特的大合集。所有外围设备的数据输入和输出都通过主内存完成，同样是以一系列0和1的形式。中央处理器像一个操作员一样处理内存中的数据，它从内存读取指令和数据，然后将运算结果写回内存。

在我们谈论内存、进程、内核和其他内容时，你会经常看到状态（state）这个词。严格说来，一个状态就是一组特定排列的比特。例如，内存中0110、0001和1011这三组比特值即表示三个不同的状态。

一个进程动辄由几百万个比特值组成，因而使用抽象词汇来描述状态可能比使用比特值更简单一些。我们可以使用进程已经完成的任务或者当前正在执行的任务来描述其状态，如“进程正在等待用户输入”或者“进程正在执行启动任务的第二个阶段”。

注解　我们通常使用抽象词汇而非比特值来描述状态，映像（image）这个词用来表示比特值在内存中的特定物理排列。

1.3　内核

我们之所以介绍主内存和状态，是因为内核的几乎所有操作都和主内存相关。其中之一是将内存划分为很多区块，并且一直维护着这些区块的状态信息。每一个进程拥有自己的内存区块，且内核必须确保每个进程只使用它自己的内存区块。

内核负责管理以下四个方面。

- 进程：内核决定哪个进程可以使用CPU。
- 内存：内核管理所有的内存，为进程分配内存，管理进程间的共享内存以及空闲内存。
- 设备驱动程序：作为硬件系统（如磁盘）和进程之间的接口，内核负责操控硬件设备。

- 系统调用和支持：进程通常使用系统调用和内核进行通信。

下面我们详细介绍一下这四个方面。

注解 如果你对内核的详细工作原理感兴趣，可以参考Abraham Silberschalz、Peter B. Galvin和Greg Gagne所著*Operating System Concepts*, 9th Edition（Wiley，2012），以及Andrew S. Tanenbaum和Herbert Bos所著*Modern Operating Systems*, 4th Edition（Prentice Hall，2014）这两本书。

1.3.1　进程管理

进程管理涉及进程的启动、暂停、恢复和终止。启动和终止进程比较直观，但是要解释清楚进程在执行过程中如何使用CPU则相对复杂一些。

在现代操作系统中，很多进程貌似都是“同时”运行的。例如，你可以同时在桌面打开Web浏览器和电子表格应用程序。然而，虽然它们表面上看是同时运行，但实际上这些应用程序背后的进程并不完全是同时运行的。

我们设想一下，在只有一个CPU的计算机系统中，可能会有很多进程可以使用CPU，但是在任何一个特定的时间段内只能有一个进程可以使用CPU。所以实际上是多个进程轮流使用CPU，每个进程使用一段时间后就暂停，然后让另一个进程使用，依次轮流，时间单位是毫秒级。一个进程让出CPU使用权给另一个进程称为上下文切换（context switch）。

进程在其时间段内有足够的时间完成主要的计算工作（实际上，进程通常在单个时间段内就能完成它的工作）。由于时间段非常短，短到我们根本察觉不到，所以在我们看来，系统是在同时运行多个进程（我们称之为多任务执行）。

内核负责上下文切换。我们来看看下面的场景，以便理解它的工作原理。

(1) CPU为每个进程计时，到时即停止进程，并切换至内核模式，由内核接管CPU控制权。

(2) 内核记录下当前CPU和内存的状态信息，这些信息在恢复被停止的进程时需要用到。

(3) 内核执行上一个时间段内的任务（如从输入输出设备获得数据，磁盘读写操作等）。

(4) 内核准备执行下一个进程，从准备就绪的进程中选择一个执行。

(5) 内核为新进程准备CPU和内存。

(6) 内核将新进程执行的时间段通知CPU。

(7) 内核将CPU切换至用户模式，将CPU控制权移交给新进程。

上下文切换回答了一个十分重要的问题，即内核是在什么时候运行的。答案就是，内核是在上下文切换时的时间段间隙中运行的。

在多CPU系统中，情况要稍微复杂一些。如果新进程将在另一个CPU上运行，内核就不需要让出当前CPU的使用权。不过为了将所有CPU的使用效率最大化，内核会使用一些其他的方式来获取CPU控制权。

1.3.2 内存管理

内核在上下文切换过程中管理内存,这是一项十分复杂的工作,因为内核要保证以下所有条件:

- 内核需要自己的专有内存空间，其他的用户进程无法访问；
- 每个用户进程有自己的专有内存空间；
- 一个进程不能访问另一个进程的专有内存空间；
- 用户进程之间可以共享内存；
- 用户进程的某些内存空间可以是只读的；
- 通过使用磁盘交换，系统可以使用比实际内存容量更多的内存空间。

新型的CPU提供了MMU（Memory Management Unit，内存管理单元），MMU使用了一种叫作*虚拟内存*的内存访问机制，即进程不是直接访问内存的实际物理地址，而是通过内核使得进程看起来可以使用整个系统的内存。当进程访问内存的时候，MMU截获访问请求，然后通过内存映射表将要访问的内存地址转换为实际的物理地址。内核需要初始化、维护和更新这个地址映射表。例如，在上下文切换时，内核将内存映射表从被移出进程转给被移入进程使用。

注解 *内存地址映射通过内存页面表（page table）来实现。*

关于内存性能，我们将在第8章详细介绍。

1.3.3 设备驱动程序和设备管理

对于设备来说，内核的角色比较简单。通常设备只能在内核模式中被访问（例如用户进程请求内核关闭系统电源），因为设备访问不当有可能会让系统崩溃。另一个原因是不同设备之间没有一个统一的编程接口，即使同类设备也如此，比如两个不同的网卡。所以设备驱动程序传统意义上来说是内核的一部分，它们尽可能为用户进程提供统一的接口，以简化开发人员的工作。

1.3.4 系统调用和系统支持

内核还对用户进程提供其他功能。例如，*系统调用*（system call或syscall）为进程执行一些它们不擅长或无法完成的工作。打开、读取和写文件这些操作都涉及系统调用。

`fork()`和`exec()`这两个系统调用对于我们了解进程如何启动很重要。

- `fork()`：当进程调用`fork()`时，内核创建一个和该进程几乎一模一样的副本。
- `exec()`：当进程调用`exec(program)`时，内核启动`program`来替换当前的进程。

除了init（参见第6章）以外，Linux中的所有用户进程都是通过`fork()`来启动的。除了创建现有进程的副本外，大多数情况下你还可以使用`exec()`来启动新的进程。一个简单的例子是你在命令行运行`ls`命令来显示目录内容。当你在终端窗口中输入`ls`时，终端窗口中的shell调用`fork()`创建一个shell的副本，然后该副本调用`exec(ls)`来运行`ls`。图1-2显示启动`ls`这样的命令时进程和系

统调用的流程。

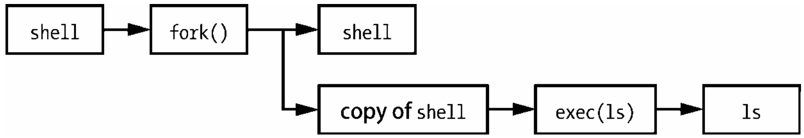


图1-2　新进程的启动

注解　系统调用通常使用括号来标记。图1-2中，进程请求内核使用fork()系统调用创建一个新的进程。这样的标记来源于C编程语言。阅读本书你不需要有C语言的知识，只需要记住系统调用是进程和内核之间的交互方式。此外，本书中我们简化了很多系统调用。例如exec()实际上是一系列具有相似功能的系统调用，只是代码实现有所不同。

除了传统的系统调用，内核还为用户进程提供其他很多功能，最为常见的是虚拟设备。虚拟设备对于用户进程而言是物理设备，但其实它们都是通过软件实现的。因此从技术角度来说，它们并不需要存在于内核中，但是实际上它们很多都存在于内核中。例如：内核的随机数生成器（/dev/random）这样的虚拟设备，如果由用户进程来实现，难度要大很多。

注解　从技术上说，用户进程还是需要通过使用系统调用打开设备的方式来访问虚拟设备，所以进程总是避免不了要和系统调用打交道。

1.4　用户空间

前面提到过，内核分配给用户进程的内存我们称之为用户空间。因为一个进程简单说就是内存中的一个状态。用户空间也可以指所有用户进程占用的所有内存。（用户空间还有一个不太正式的名称，叫userland。）

Linux中大部分的操作都发生在用户空间中。虽然从内核的角度来说所有进程都是一样的，但是实际上它们执行的是不同的任务。相对于系统组件，用户进程位于一个基础服务层中。图1-3就展示了一组组件在Linux系统中是如何交互工作的。其中最底层是基础服务层，工具服务在中间，用户使用的应用程序在最上层。图1-3是一个简化版本，你可以看到顶层距离用户最近（如用户接口和Web浏览器）。中间一层中有邮件服务器这样的组件供Web浏览器使用。最下层是一些更小的服务组件。

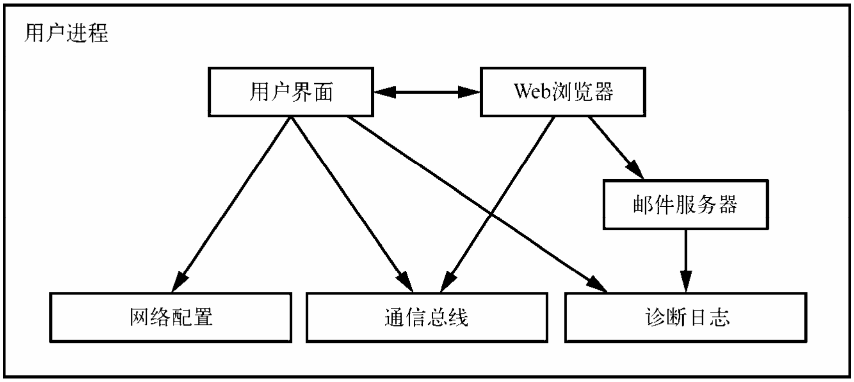


图1-3 进程类型和相互间的交互

最下层通常是由一些小的组件组成，它们比较精巧，专注完成某一个特定功能。中间层的组件比较大一些，如邮件、打印和数据库服务。顶层组件完成用户交互和复杂的功能。组件之间也可以相互调用。如果组件A调用了组件B的功能，我们可以视为组件A和B在同一层级，或者B在A之下。

然而，图1-3只是一个粗略图，实际上用户空间里没有很明显的界限。例如许多应用程序和服务会将系统诊断信息写入*日志*，大部分程序使用标准的系统日志服务来完成，但也有一些程序是自己实现日志功能。

此外，很多用户空间组件比较难分类，像Web服务器和数据库服务器这样的服务组件，你可以认为它们在图1-3中属于高级别组件，因为它们复杂度很高。然而用户应用程序也会经常调用它们的功能，所以你也可以将它们归入中级别组件。

1.5 用户

Linux内核支持用户这一Unix的传统概念。一个*用户*代表一个实体，它有权限运行用户进程，对文件拥有所有权。每个用户都有一个*用户名*，如billyjoe。然而内核是通过*用户*ID来管理用户的，用户ID是一串数字标识（详见第7章）。

用户机制主要用于权限管理。每一个用户进程都有一个用户作为*所有者*，我们称其为以该用户运行的进程。在一定限制条件下，用户可以终止和改变他的进程的行为。但是对其他用户的进程无权干预。此外，用户可以决定是否将属于自己的文件和其他用户共享。

Linux操作系统的用户包括系统自带用户和供人使用的用户。详情见第3章。其中最关键的用户是root*用户*（意思是根用户或超级用户）。root用户不受前面提到的种种权限的限制，它可以终止其他用户的进程，读取系统中的任何文件。因此root也被称作超级用户。Unix的系统管理员拥有超级用户权限。

注解　使用root权限操作系统是一件很危险的事情，因为用户拥有最高权限，可以为所欲为，一旦出错很难定位和恢复。因此系统管理员通常尽量避免使用root权限。而且，root用户虽然权限很高，但是还是在用户模式而非内核模式中运行。

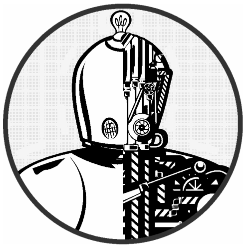
用户组是指一组用户的集合。用户组的主要作用是允许一个用户同组内的其他用户共享文件权限。

1.6　前瞻

至此我们对Linux系统的组成有了一个大致的了解。用户和用户进程交互，内核管理进程和硬件系统。内核和进程都在内存中运行。

这些基础知识固然很重要，但如果想要了解更多的细节，你需要实际操作一番。下一章你会了解到一些用户空间的基础知识，还有本章没有提及的永久存储（硬盘、文件等），就是存放应用程序和数据的地方。

第 2 章 基础命令和目录结构



本章我们将介绍Unix系统的命令和工具，它们在本书中会经常被用到。你可能已经对这些基本知识有所了解，不过我还是建议你花些时间再阅读一遍，特别是2.19节关于目录结构的阐述。

你也许会问，为什么要介绍Unix命令？这本书不是关于Linux的吗？没错，Linux其实是Unix的一个变种，它的本质还是Unix。Unix这个词在本章中出现的频率甚至高于Linux，并且你可以将本章的知识直接应用到其他基于Unix的操作系统，如Solaris和BSD。我们尽量避免介绍太多Linux特有的内容，一方面可以让你多了解一点其他的操作系统，另一方面也因为那些只对Linux适用的扩展功能往往不太稳定可靠。掌握核心命令能够让你很快上手任何新的基于Linux的操作系统。

注解 Unix初学者若想了解更多细节，可以参考这几本书：*The Linux Command Line*（No Starch Press，2012）、*UNIX for the Impatient*（Addison-Wesley Professional，1995）和*Learning the UNIX Operating System*, 5th edition（O'Reilly，2001）。

2.1 Bourne shell: /bin/sh

shell意思为命令行界面，是Unix操作系统中最为重要的部分之一。shell是运行命令行的应用程序，而命令行就是用户输入的那些命令。同时它为Unix程序员提供了一个小的编程环境，在这里Unix程序员可以将通用的任务分解为一些小的组件，然后使用shell来管理和组织它们。

Unix操作系统中很多重要的部分其实都是shell脚本，它们是包含一系列shell命令的文本文件。如果你用过MS-DOS，你可以将shell脚本理解为功能强大的.bat批处理文件。我们将在第11章详细介绍shell脚本。

通过本书的阅读和练习，你将会逐渐熟练地使用shell来运行各种命令。它的一个好处是一旦出现了误操作，你可以清楚地看到你的输入错误，然后进行修正。

Unix的shell有很多种，它们都是基于Bourne shell（/bin/sh）这个贝尔实验室开发的标准shell，在早期的Unix系统上运行。所有基于Unix的操作系统都需要Bourne shell才能正常工作。

Linux使用了一个增强版本的Bourne shell，我们称之为bash或者“Bourne-again” shell。大部分Linux系统的默认shell是bash，其通常有一个符号链接/bin/sh。你需要使用bash来运行本书中的例子。

注解　你的Unix系统管理员为你设置的默认shell可能不是bash，你可以使用chsh命令来更改，或者请管理员为你更改。

2.2　shell 的使用

安装Linux时，除了默认的root账号外，你还需要为自己创建至少一个普通用户账号，这些账号将会是你的个人账号。本章中你需要使用普通用户账号。

2.2.1　shell窗口

登录系统后，打开一个shell窗口（也叫作终端窗口）。打开shell窗口最简单的方法是，在Gnome或者Ubuntu Unity这样的图形用户界面（Graphical User Interface，以下简称GUI）中运行终端程序，这样就可以在新的窗口中启动shell。通常在窗口的顶端你能看到一个$提示符。在Ubuntu上，提示符是这样：`name@host:path$`（用户名@主机名:路径$）。在Fedora上，提示符是这样：`[name@host path]$`。shell窗口类似Windows上的DOS，OS X系统上的终端程序本质上和Linux中的shell窗口一样。

本书中的很多命令都可以在shell上运行，例如你可以输入以下命令行（不用输入前面的$），然后按回车键：

```
$ echo Hello there.
```

注解　本书中许多shell命令都以#开头，需要以root身份来运行，运行时需要格外小心。

现在试试下面这个命令：

```
$ cat /etc/passwd
```

这个命令是将文件/etc/passwd中的内容显示到shell窗口中。有关这个文件的内容我们会在第7章详细介绍。

2.2.2　cat命令

cat命令很简单，它显示一个或者多个文件的内容，命令语法如下：

```
$ cat file1 file2 ...
```

上面这个cat命令会显示file1和file2等文件的内容，然后退出。之所以叫cat是因为如果有多个文件的话，它会把这些文件的内容拼接起来显示。

2.2.3　标准输入输出

我们将使用cat命令来学习Unix的输入和输出（以下简称I/O）。Unix进程使用I/O流来读写数据。进程从输入流中读取数据，向输出流写出数据。数据流非常灵活，比如输入流可以是文件、设备、终端，甚至还可以是来自其他进程的输出流。

想知道输入流的工作原理，只需要输入cat命令并回车，这时候你会看到屏幕上没有显示任何结果，因为cat命令仍在运行中。现在你输入几个字符然后回车，你会看到cat命令会在屏幕上显示出你刚刚输入的字符。最后你可以在任意空白行按CTRL-D终止cat命令的执行并回到shell提示符。

你刚刚和cat命令进行的一系列交互就是通过数据流机制来实现的。因为你没有指定输入文件名，cat命令就从Linux内核提供的默认标准输入流中获得输入数据，这时运行cat命令的终端就成为标准输入。

注解　按CTRL-D终止当前终端的标准输入并终止命令（通常会终止一个程序）。这和CTRL-C不一样。CTRL-C是终止当前进程的运行，无论是否有输入和输出。

标准输出也与之类似。内核为每个进程提供一个标准输出流供它们输出数据。cat命令在终端运行的时候，标准输出就和该终端建立连接，cat命令将数据输出到标准输出，就是你在屏幕上看到的结果。

标准输入和标准输出通常简写为stdin和stdout。很多命令和cat一样，如果你不为它们指定输入文件，他们就从标准输入获得数据。输出则有点不同，一部分命令（如cat）将数据输出到标准输出，另一部分命令可以将数据直接输出到文件。

除了标准输入和输出外，还有标准错误信息流，我们将在2.14.1节介绍。

标准流的一个优点是你可以随心所欲地指定数据的输入输出来源，在2.14节中我们会介绍如何将流连接到文件和其他进程。

2.3　基础命令

本节将介绍更多的Unix命令。它们大都需要输入参数，同时支持可选项和格式（由于数量太

多，在此不一一列出）。下面是一些基础命令的简单介绍，我们暂不深入讲解。

2.3.1　ls命令

ls命令显示指定目录的内容，默认参数为当前目录。ls -l显示详细的列表，ls -F显示文件类型信息（文件类型和权限将在2.17节介绍）。下面是文件详细列表的一个示例，其中第三列是文件的所有者，第四列是用户组，第五列是文件大小，后面是文件更改的时间、日期以及文件名。

```
$ ls -l
total 3616
-rw-r--r--   1 juser    users       3804 Apr 30  2011 abusive.c
-rw-r--r--   1 juser    users       4165 May 26  2010 battery.zip
-rw-r--r--   1 juser    users     131219 Oct 26  2012 beav_1.40-13.tar.gz
-rw-r--r--   1 juser    users       6255 May 30  2010 country.c
drwxr-xr-x   2 juser    users       4096 Jul 17 20:00 cs335
-rwxr-xr-x   1 juser    users       7108 Feb  2  2011 dhry
-rw-r--r--   1 juser    users      11309 Oct 20  2010 dhry.c
-rw-r--r--   1 juser    users         56 Oct  6  2012 doit
drwxr-xr-x   6 juser    users       4096 Feb 20 13:51 dw
drwxr-xr-x   3 juser    users       4096 May  2  2011 hough-stuff
```

第一列中的d我们将在2.17节详细介绍。

2.3.2　cp命令

cp命令用来复制文件。下面的命令将文件file1复制到文件file2：

```
$ cp file1 file2
```

下面的命令将多个文件（file1 ... fileN）复制到目录dir：

```
$ cp file1 ... fileN dir
```

2.3.3　mv命令

mv命令有点类似cp，用来重命名文件。下面的命令将文件名从file1重命名为file2：

```
$ mv file1 file2
```

你也可以使用mv将多个文件移动到某个目录：

```
$ mv file1 ... fileN dir
```

2.3.4　touch命令

touch命令用来创建文件。如果文件已经存在，则该命令会更新文件的时间戳，就是我们在ls -l命令的执行结果中看到的文件更新时间和日期。下面的命令创建一个新的文件，内容为空：

```
$ touch file
```

如果我们对文件执行ls -l，你将会看到下面的显示结果，其中❶就是文件被创建的时间和日期：

```
$ ls -l file
-rw-r--r-- 1 juser users 0 May 21 18:32❶ file
```

2.3.5　rm命令

rm命令用来删除文件，文件一旦被删除通常无法恢复：

```
$ rm file
```

2.3.6　echo命令

echo命令将它的参数显示到标准输出，例如：

```
$ echo Hello again.
Hello again.
```

我们在查看shell通配符展开（如*这样的通配符）和环境变量（如$HOME）的时候经常使用echo命令，本章稍后会详细介绍。

2.4　浏览目录

Unix的目录结构是从/开始，有时候也叫作root目录。目录之间使用斜杠/分隔，而不是Windows中的反斜杠\。root目录/下有子目录，如/usr，详见2.19节。

我们通过路径或路径名来访问文件。以/开头的路径（如/usr/lib）叫绝对路径。

两个点（..）代表一个目录的上层目录。如果你当前在目录/usr/lib中，那..就代表/usr目录，../bin则代表/usr/bin。

一个点（.）代表当前目录。如果你当前在/usr/lib目录中，.就代表/usr/lib，./X11则代表/usr/lib/X11。通常我们不需要使用.，而是直接使用目录名来访问当前目录下的子目录，如X11效果和./X11一样。

不以/开头的路径叫相对路径，我们大部分时候都基于当前所在目录使用相对路径。下面介绍一些和目录操作相关的命令。

2.4.1　cd命令

cd命令用来设置当前工作目录。当前工作目录是指你的进程和shell当前所在的目录。

```
$ cd dir
```

如果不带dir参数，cd命令会返回你的个人主目录，指的是你登录系统后进入的目录。

2.4.2　mkdir命令

mkdir命令用来创建新目录，例如，下面的命令创建一个名为dir的新目录：

```
$ mkdir dir
```

2.4.3　rmdir命令

rmdir命令用来删除目录：

```
$ rmdir dir
```

如果要删除的目录里面有内容（文件和其他目录），上面的命令会执行失败。因为rmdir只能删除空目录，你可以使用rm -rf来删除一个目录以及其中的所有内容。使用这个命令的时候要非常小心，尤其是当你是超级用户（root或superuser）的时候。因为-r选项会依次删除dir中的所有文件和子目录，-f选项代表强制删除。所以使用-rf时尽量不要在参数里使用通配符（如*），并且执行命令前最好检查参数是否正确。

2.4.4　shell通配符

shell可以使用通配符来匹配文件名和目录名。其他的操作系统也有通配符这个概念。比如*代表任意字符和数字。下面的命令列出当前目录中的所有文件：

```
$ echo *
```

shell根据参数中的通配符来匹配文件名。shell将命令中的参数替换为实际的文件名，这个过程我们称为*展开*。比如：

- at*展开为所有以at开头的文件名；
- *at展开为所有以at结尾的文件名；

- *at*展开为所有包含at的文件名。

如果通配符没有匹配的文件名，shell就不进行任何的展开，参数按照原样来执行，比如：echo *dfkdsafh。

注解 如果你惯于使用MS-DOS，你可能会下意识地使用*.*来匹配所有文件。在Linux系统和其他Unix系统中，*.*只匹配那些包含.的文件名和目录名，而Unix系统中很多文件名是没有的。

另外一个shell通配符问号（?）帮助shell确切匹配任意一个字符，如b?at与boat和brat相匹配。

如果不想让shell展开通配符，你可以使用单引号（''）。例如运行echo '*'将会显示一个*。在一些命令如grep和find中，这样做非常有用（这一内容将在11.2节详细介绍）。

注解 需要注意的是，shell是先展开通配符，然后执行命令行。如果*传递到命令行的时候仍然未能展开，shell则对此无能为力，一切都取决于命令本身如何处理。

现代shell的模式匹配能力并不仅限于此，但*和?这两种是你必须要掌握的。

2.5 中间命令

下面我们介绍一些基本的Unix中间命令。

2.5.1 grep命令

grep命令显示文件和输入流中和参数匹配的行。如下面的命令显示文件/etc/passwd中包含文本root的所有行：

```
$ grep root /etc/passwd
```

在对多个文件进行批量操作的时候，grep命令非常好用，因为它显示文件名和匹配的内容。如果你想查看目录/etc中所有包含root的文件，可以执行以下命令：

```
$ grep root /etc/*
```

grep命令有两个比较重要的选项，一个是-i（不区分大小写），一个是-v（反转匹配，就是显示所有不匹配的行）。grep还有一个功能强大的变种叫作egrep（实际上就是grep -E）。

grep命令能够识别正则表达式。正则表达式比通配符功能更强大，下面是两个例子：

- .*匹配任意多个字符（类似*通配符）；
- .匹配任意一个字符。

> **注解**　帮助手册grep(1)中有关于正则表达式的详细说明，不过对于读者来说可能比较不方便理解。你可以参考这两本书：*Mastering Regular Expression*, 3rd edition（O’Reilly，2006）或者*Programming Perl*, 4th edition（O’Reilly，2012）中的“the regular expression”一章。如果你对数学和正则表达式的历史感兴趣，可以参阅*Introduction to Automata Theory, Language, and Computation*, 3rd edition（Prentice Hall，2006）。

2.5.2　less命令

当要查看的文件过大或者内容多得需要滚动屏幕的时候，可以使用less命令。如要查看像/usr/share/dict/words这样的大文件，可以使用less /usr/share/dict/words命令。less命令可以将内容分屏显示，按空格键可查看下一屏，B键查看上一屏，Q键退出。

> **注解**　less命令实际上是more命令的增强版本。绝大多数Linux系统中都有这个命令，但是一些Unix系统和嵌入式系统中没有这个命令，这时你可以使用more命令。

你可以在less命令的输出结果中进行搜索。例如：使用/word从当前位置向前搜索word这个词，使用?word从当前位置向后搜索。当找到一个匹配的时候，按N键可以跳到下一个匹配。

你可以将几乎所有进程的输出作为另一个进程的输入，我们将在2.14节详细介绍。当你执行的命令涉及很多输出，或者你想使用less来查看输出结果的时候，这个方法非常管用，比如下例所示：

```
$ grep ie /usr/share/dict/words | less
```

你可以自己亲身实践一下这个命令。类似这样的less代码你会常用到。

2.5.3　pwd命令

pwd命令仅输出当前的工作目录名。这个命令看上去不是那么有用，其实不然，它有以下两个用处。

首先，并不是所有的提示符都显示当前目录名，甚至有时候你需要摆脱它，因为它占用很大空间，这时候就需要使用pwd来解决。

其次，使用符号链接（我们将在2.17.2节介绍）的时候通常很难获知当前目录信息，这时我们可以使用pwd -P来查看。

2.5.4　diff命令

diff命令用来查看两个文件之间的不同，例如：

```
$ diff file1 file2
```

该命令有几个选项可以让你设置输出结果的格式，不过默认的格式对于我们来说已经足够清晰易读了。很多开发人员喜欢用diff -u格式，因为这个格式能被许多自动化工具很好地识别。

2.5.5 file命令

如果你想知道一个文件的格式信息，可以执行file命令：

```
$ file file
```

这个看似平淡无奇的命令会给你提供很多有用的信息。

2.5.6 find和locate命令

我们有时候会碰到一种让人抓狂的情况，就是明明知道有那么一个文件，但就是不知道它在哪个目录。别急，使用find命令可以帮你在目录中寻找文件：

```
$ find dir -name file -print
```

find命令能做很多事情，但是在你确定你了解-name和-print选项之前，不要尝试诸如-exec这样的选项。find命令可以使用模式匹配参数（如*），但是必须加引号（'*'），以免shell自动将它们展开。（回想2.4.4节讲的，shell在运行命令前会展开通配符。）

另外一个查找文件的命令是locate。和find不同的是，locate在系统创建的文件索引中查找文件。这个索引由操作系统周期性地进行更新，查找速度比find更快。但是locate对于查找新创建的文件可能会无能为力，因为它们有可能还没有被加入到索引中。

2.5.7 head和tail命令

head命令显示文件的前10行内容（例如head /etc/passwd）。tail命令显示文件的最后10行内容（如tail /etc/passwd）。

你可以使用-n选项来设置显示的行数（例如：head -5 /etc/passwd）。如果要从第n行开始显示所有内容，使用tail +n。

2.5.8 sort命令

sort命令将文件内的所有行按照字母顺序快速排序。你可以使用-n选项按照数字顺序排序那些以数字开头的行。使用-r选项反向排序。

2.6　更改密码和 shell

你可以使用passwd命令来更改密码，你需要输入一遍你的旧密码和两遍新密码。密码最好复杂一些，不要使用简单的词句，最好是数字、大小写字母和特殊字符混合。

设置密码的一个好方法是选择一个你能记住的短句，将其中的某些字符替换为数字和标点，然后将这个密码记牢。

你可以用chsh命令更改shell（如改为ksh或tcsh）。本书默认使用的shell是bash。

2.7　dot 文件

现在跳转到你的home目录，分别运行ls和ls -a两个命令，你应该能够注意到一些区别。如果没有-a选项，你无法看到那些叫作dot文件的配置文件，这些文件以.开头。常见的dot文件有.bashrc和.login，还有以.开头的dot目录，如.ssh。

这些dot文件没有什么特别之处。有些命令不显示它们是为了让你的个人主目录显得更简洁。例如，除非使用-a选项，否则ls命令不显示dot文件。此外，shell通配符不匹配dot文件，除非明确指定.*。

注解　在通配符中使用.*可能会导致一些问题，因为.*匹配.和..（当前目录和上级目录）。你可以使用正则表达式.[^.]*或.??*来排除这两个目录。

2.8　环境变量和 shell 变量

shell中可以保存一些临时变量，称作shell变量，它们是一些字符值。shell变量可以保存脚本执行过程中的数据，一些shell变量用来控制shell的运行方式（例如，bash shell在显示提示符前会读取变量PS1的值，如果PS1变量中有内容，则将它看作提示符）。

我们使用等号=为shell变量赋值，例如：

```
$ STUFF=blah
```

以上命令将blah赋值给变量STUFF。我们使用$STUFF来获得该变量的值（例如，尝试一下echo $STUFF这个命令）。我们将在第11章介绍更多的shell变量。

环境变量和shell变量类似，但其不仅仅针对shell。Unix系统中所有的进程都能够访问环境变量。两者最大的区别是shell变量只能被当前的shell访问，在shell中运行的命令则无法访问。而环境变量能够被shell中运行的所有进程访问。

环境变量可以通过export命令来设置。例如，如果想将shell变量$STUFF变成环境变量，可以执行如下命令：

```
$ STUFF=blah
$ export STUFF
```


许多程序使用环境变量作为配置和选项信息。例如，你可以使用LESS这个环境变量来配置less命令的参数（许多命令的帮助手册里都有ENVIRONMENT这一节，教你如何使用环境变量来设置该命令的参数和选项）。

2.9　命令路径

PATH是一个特殊的环境变量，它定义了命令路径，或简称为路径。命令路径是一个系统目录列表，shell在执行一个命令的时候，会去这些目录中查找这个命令。比如：运行ls命令时，shell会在PATH中定义的所有目录里查找ls，如果ls出现在多个目录中，shell会运行第一个匹配的程序。

如果你运行echo $PATH，你会看到所有的路径组件，它们之间以冒号（:）分隔。例如：

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin
```

你可以设置PATH变量，为shell查找命令加入更多的路径。例如，使用以下命令可以将路径dir加入到PATH的最前面，这样shell会先查找dir路径，然后再查找其他路径：

```
$ PATH=dir:$PATH
```

你也可以将路径加入到PATH变量的最后面，这样shell会最后查找dir路径：

```
$ PATH=$PATH:dir
```

注解　在更改PATH时需要特别小心，因为你有可能会不小心将PATH中所有的路径删除掉。不过也不用太担心，你只需要启动一个新的shell就可以找回原来的PATH。最简单的解决办法是关闭当前的终端窗口并启动一个新的窗口。

2.10　特殊字符

在谈论Linux的时候，我们需要了解一些术语。如果你有兴趣了解，可参考“Jargon File”（http://www.catb.org/jargon/html/）或者它的印刷版本*The New Hacker's Dictionary*（MIT Press，1996）。

表2-1列出了一些特殊字符，其中很多本章已经介绍过。一些工具，比如Perl编程语言，用到几乎所有这些特殊字符！（请注意这里使用的字符名称是美国英语名称。）

表2-1　特殊字符

字　　符	名　　称	用　　途
*	星号	正则表达式，通用字符
.	句点	当前目录，文件/主机名的分隔符
!	感叹号	逻辑非运算符，命令历史
|	管道	命令管道
/	斜线	目录分隔符，搜索命令
\	反斜线	常量，宏（非目录）
$	美元符号	变量符号，行尾
'	单引号	字符串常量
`	反引号	命令替换
"	双引号	半字符串常量
^	脱字符	逻辑非运算符，行头
~	波浪字符	逻辑非运算符，目录快捷方式
#	井号	注释，预处理，替换
[ ]	方括号	范围
{ }	大括号	声明块，范围
_	下划线	空格的简易替代

注解　控制键我们通常用^来表示，如^C代表CTRL-C。

2.11　命令行编辑

在使用shell时，你应该能注意到可以使用左右箭头来编辑命令行，并且通过上下箭头来查看之前的命令。这是Linux系统的标准操作。

但使用ctrl键来代替箭头键会更加方便。表2-2中的命令是Unix系统的文本编辑标准命令，掌握了这些，你就可以很方便地在任何Unix系统中编辑文本。

表2-2　命令行按键

按　　键	操　　作
CTRL-B	左移光标
CTRL-F	右移光标
CTRL-P	查看上一条命令（或上移光标）
CTRL-N	查看下一条命令（或下移光标）
CTRL-A	移动光标至行首
CTRL-E	移动光标至行尾
CTRL-W	删除前一个词
CTRL-U	删除从光标至行首的内容

（续）

按　键	操　作
CTRL-K	删除从光标至行尾的内容
CTRL-Y	粘贴已删除的文本（例如粘贴CTRL-U所删除的内容）

2

2.12　文本编辑器

说到文本编辑，不得不提文本编辑器。要用好Unix，你必须能够编辑文本文件并且不对其造成损坏。Unix系统使用纯文本文件来保存配置信息（如目录/etc中的文件）。编辑文件并不是难事，但由于要经常性地编辑这些文件，因此你需要一个强大的文本编辑器。

在众多文本编辑器中，你需要掌握vi和Emacs二者之一，它们是Unix系统中约定俗成所用的标准编辑器。很多Unix的配置向导对编辑器的选择很挑剔，但是没关系，你可以自己选择，选择标准就是它对你而言合不合适。

- 如果你想要一个万能的编辑器，功能强大，有在线帮助，你可以试试Emacs。不过它需要你进行一些额外的手动编辑。
- 如果想要高效快速，那么vi比较适合你。

有关vi的详细知识可以参考这本书：*Learning the vi and Vim Editors: Unix Text Processing*, 7th edition（O'Reilly，2008）。关于Emacs你可以参考在线文档Start Emacs：打开Emacs，按CTRL-H，然后按T， 或者参考*GNU Emacs Manual*（Free Software Foundation，2011）。

其他的编辑器如Pico和myriad GUI editor可能界面会更加友好一些，但是你一旦习惯了vi和Emacs以后，也许就再也不想使用它们了。

注解　你在进行文本编辑的时候，可能第一次注意到了终端界面和GUI的区别。vi这样的编辑器运行在终端窗口中，使用标准输入输出界面。GUI编辑器启动自己的窗口并有自己的窗口界面，与终端窗口是相互独立的。Emacs既有终端界面也有图形界面。

2.13　获取在线帮助

Linux系统的帮助文档非常丰富。帮助手册提供命令的使用说明。比如你若是想了解ls命令的用法，只需运行：

```
$ man ls
```

帮助手册旨在提供基础知识和参考信息，有时会有一些实例和交叉索引，但是基本没有那种教程式的文档。

帮助手册会按系统排序方式（如按照字母顺序）列出命令的所有选项，但是不会突出重点（比

如那些经常被使用的选项)。如果你有足够的耐性，可以逐个尝试，或者可以问别人。

下面的命令可以帮你借助关键字来查找相关帮助手册：

```
$ man -k keyword
```

如果你只知道某个功能，但是不知道命令名，你可以很方便地通过关键字来查找。比如你若想使用排序功能，就可以运行下面的命令来列出所有和排序有关的命令：

```
$ man -k sort
--snip--
comm (1)            - compare two sorted files line by line
qsort (3)           - sorts an array
sort (1)            - sort lines of text files
sortm (1)           - sort messages
tsort (1)           - perform topological sort
--snip--
```

输出结果包括帮助手册的名称、所属的章节以及内容的简要描述。

注解　如果你对本书目前介绍的命令有疑问，可以使用man命令查阅它们的帮助手册。

帮助手册按照命令类型被组织为很多个章节，章节编号出现在章节名后面的括号中，例如ping(8)。表2-3中列出了各章节和它们的编号。

表2-3　在线帮助手册章节列表

章　节	简　介
1	用户命令
2	系统调用
3	Unix高级编程库文档
4	设备接口和设备驱动程序信息
5	文件描述符（系统配置文件）
6	游戏
7	文件格式、规范和编码（ASCII编码和文件后缀等等）
8	系统命令和服务器

章节1、5、7和8对本书的内容是很好的补充参考。章节4用到的不多，章节6的内容稍微有些单薄。章节3主要是供开发人员参考。在阅读完本书有关系统调用的部分后，你能对章节2的内容有更好的理解。

你可以按序号来选择章节，这会让搜索结果更加精确，因为一旦匹配了搜索关键字，帮助手册会定位到该关键字查找结果的第一页。比如你要搜索有关passwd的信息，可以使用如下命令：

```
$ man 5 passwd
```

帮助手册涵盖的是基本内容，你还可以使用--help或者-h选项来获得帮助信息。如ls --help。

GNU项目因为不喜欢帮助手册这种方式，引入了info（或者texinfo）。info文档的内容更加丰富，同时也更复杂一些。可以使用info命令查看info文件内容：

```
$ info command
```

有一些程序将它们的文档放到目录/usr/share/doc中，而不是man和info里。你可以在这里搜索需要的文档，当然别忘了还有互联网。

2.14 shell 输入输出

至此你已经了解了Unix的基本命令，文件和目录，现在我们可以介绍标准输入输出的重定向了。让我们从标准输出开始。

如果想将命令的执行结果输出到文件（默认是终端屏幕），可以使用重定向字符>：

```
$ command > file
```

如果文件file不存在，shell会创建一个新文件file。如果file文件已经存在，shell会先清空文件的内容。（一些shell可以通过设置参数来防止文件被清空，如：bash中的set -C命令。）

如果不想把原文件覆盖，你可以使用>>将命令的输出结果加入到文件末尾：

```
$ command >> file
```

这个方法在收集多个命令的执行结果时非常有用。

你还可以使用管道字符（|）将一个命令的执行结果输出到另一个命令。例如：

```
$ head /proc/cpuinfo
$ head /proc/cpuinfo | tr a-z A-Z
```

你可以使用任意多个管道字符，只需要在每个额外的命令前各加一个管道符即可。

2.14.1 标准错误输出

有的时候你会发现，即使重定向了标准输出，终端屏幕上还是会显示一些信息，其实这是标准错误输出，是用来显示系统错误和调试信息的一种额外的输出流。比如，运行下面的命令后，会发生错误：

```
$ ls /ffffffffff > f
```

输出完成后，f应该是空的，但你会在终端屏幕上看到以下错误信息，即标准错误输出：

```
ls: cannot access /fffffffff: No such file or directory
```

如果有必要，你可以使用2>重定向标准错误输出，例如，使用2>向f发送标准输出，向e发送标准错误输出：

```
$ ls /fffffffff > f 2> e
```

这里的2是由shell修改的流ID，1是标准输出，2是标准错误输出。

你也可以使用>&将标准输出和标准错误输出重定向到同一个地方，例如，把标准输出和标准错误输出重定向到文件f中，可执行以下命令：

```
$ ls /fffffffff > f 2>&1
```

2.14.2　标准输入重定向

使用<操作符将文件内容重定向为命令的标准输入：

```
$ head < /proc/cpuinfo
```

你偶尔会遇见要求这种类型的重定向的程序，但因为很多Unix命令可以使用文件名作为参数，所以不太常需要使用<来重定向文件。例如，上述命令也可以写成`head /proc/cpuinfo`。

2.15　理解错误信息

在Linux这样的基于Unix的操作系统中，程序运行出错时你要做的第一件事情必须是查看错误信息，因为大多数情况下，出错的具体原因都能在错误信息里找到。这一点Unix系统做得比其他有些操作系统要好。

2.15.1　解析Unix的错误信息

绝大部分Unix上的应用程序都使用相同的方式处理错误信息，但在任意两个程序的输出结果之间可能会存在些微的差别。例如，你可能会经常遇到这种情况：

```
$ ls /dsafsda
ls: cannot access /dsafsda: No such file or directory
```

该信息分为以下三部分。

- 命令文件名：`ls`。一些程序不显示命令文件名，这对于脚本调试来说很不方便。但这也不是问题的关键。

- 文件路径：/dsafsda。这是一条更为具体的信息，而问题就出在这个文件路径上。
- 错误信息：No such file or directory。这一信息告诉我们错误出在文件路径名上。

将以上的信息综合起来看，你就能得出结论：ls想要访问文件/dsafsda，但是文件不存在。在这个例子里，错误信息很易懂，但是如果你运行的是执行很多命令的脚本，出错信息会变得复杂难懂。

在排错时，务必从第一个错误开始入手。程序报告错误时总是先告诉你它无法完成某一个操作，接下来告诉你一些其他的相关问题。例如我们虚构这样一个场景：

```
scumd: cannot access /etc/scumd/config: No such file or directory
```

后面跟着一大串的错误信息，看起来问题很严重。首先不要受它们的影响，专注于第一个错误信息你就知道，要解决的问题只不过是创建一个文件/etc/scumd/config而已。

注解　不要把错误信息和警告信息混为一谈。警告信息看起来像是错误信息，但是它只是告诉我们程序出了问题，但是还能够继续运行。要解决警告信息里面的问题，你可能需要终止当前进程。（有关查看和终止进程的内容，我们将在2.16节介绍。）

2.15.2　常见错误

Unix程序的很多错误与文件和进程有关。下面我们列举一些常见的错误。

No such file or directory

这可能是我们最常遇到的错误：访问一个不存在的文件或目录。由于Unix的I/O系统对文件和目录不做区分，所以当你试图访问一个不存在的文件，进入一个不存在的目录，或将文件写入一个不存在的目录时，都会出现这个错误。

File exists

如果新建文件的名称和现有的文件或者目录重名，就会出现这个错误。

Not a directory, Is a directory

这个错误出现在当你把文件当作目录或者反之，把目录当作文件。例如：

```
$ touch a
$ touch a/b
touch: a/b: Not a directory
```

错误出在第二个命令这里，将文件a当作了目录，很多时候你可能需要花点时间来检查文件路径。

No space left on device

说明硬盘空间不足。

Permission denied

当你试图读或写一个没有访问权限的文件或目录时，会遇到这个错误。当你试图执行一个你无权执行（即使你有读的权限）的文件时也会出现这个错误。我们会在2.17节详细介绍。

Operation not permitted

当你试图终止一个你无权终止的进程时，会出现这个错误。

Segmentation fault, Bus error

分段故障，总线错误。分段故障这个错误通常是告诉你，你运行的程序出了问题。可能你的程序试图访问它无权访问的内存空间，这时操作系统就会将其终止。总线错误说明你的程序访问内存的方式有问题。遇到这类错误通常是因为程序的输入数据有问题。

2.16 查看和操纵进程

我们在第1章介绍过，进程就是运行在内存中的程序。每个进程都有一个数字ID，叫进程ID（Process ID，以下简称PID）。可以使用ps命令列出所有正在运行的进程：

```
$ ps
  PID   TTY  STAT TIME   COMMAND
  520    p0  S    0:00   -bash
  545     ?  S    3:59   /usr/X11R6/bin/ctwm -W
  548     ?  S    0:10   xclock -geometry -0-0
 2159    pd  SW   0:00   /usr/bin/vi lib/addresses
31956    p3  R    0:00   ps
```

每行的字段依次代表以下内容。

PID：进程ID。

TTY：进程所在的终端设备，稍后详述。

STAT：进程状态，就是进程在内存中的状态。例如，S表示进程正在休眠，R表示进程正在运行。（完整的状态列表请参阅帮助手册ps(1)。）

TIME：进程目前为止所用CPU时长（格式：mm:ss），就是进程占用CPU的总时长。

COMMAND：命令名，请注意进程有可能将其由初始值改为其他。

2.16.1 命令选项

ps命令有很多选项，你可以使用三种方式来设置选项：Unix方式、BSD方式和GNU方式。BSD方式被认为是比较好的一种，因为它相对简单一些。本书也将使用BSD这种方式。下面是一些比较实用的选项组合。

ps x 显示当前用户运行的所有进程。

ps ax 显示系统当前运行的所有进程，包括其他用户的进程。

ps u 显示更详细的进程信息。

ps w 显示命令的全名，而非仅显示一行以内的内容。

和对其他程序一样，你可以对ps使用选项组合，如ps aux和ps auxw。你可以将PID作为ps命令的一个参数，用来查看该特定进程的信息，如ps u $$，其中$$是一个shell变量，表示当前的shell进程。（在第8章我们将会介绍top和lsof管理员命令，它们能够帮助我们找到进程所在的位置。）

2.16.2 终止进程

要终止一个进程，可以使用kill命令向其发送一个信号。当kill命令运行时，它请求内核发送一个信号给进程。大多数情况下，你可以执行下面的命令：

```
$ kill pid
```

信号的种类有很多，默认是TERM（或者terminate）。你可以设置选项来发送不同类型的信号。例如，发送STOP信号可以让进程暂停，而不是终止：

```
$ kill -STOP pid
```

被暂停的进程仍然驻留在内存，等待被继续执行。使用CONT信号可以继续执行进程：

```
$ kill -CONT pid
```

注解 你可以使用CTRL-C来终止当前运行的进程，效果和kill -INT命令一样。

终止进程最粗鲁的一种方式是使用KILL信号。和其他信号不同，KILL会强行终止进程，并将其移出内存，不会给进程清理和收尾的机会。不到万不得已最好不要使用该信号。

不要随便终止一个你不知道的进程，不然很有可能遇到麻烦。

你还可以使用数字来代替信号名，例如：kill -9等同于kill -KILL。因为内核使用数字来代表不同的信号。如果你知道你想要发送的信号的数字号，可以使用这种方式。

2.16.3 任务控制

Shell也支持任务控制（Job Control），是通过不同的按键和命令向进程发送TSTP（类似STOP）和CONT信号的一种方式。例如，你可以使用CTRL-Z发送TSTP信号来停止进程，然后键入fg（将进程置于前台）或者bg（将进程移入后台，见下一小节）继续运行进程。对初学者来说这些可能不太好理解，不过对于很多高级用户来说它们是很好用的命令。如果使用CTRL-Z而不是CTRL-C，然后置之不理，最终会形成大量处于暂停状态的进程。

提示 你可以使用jobs命令来查看你暂停了哪些进程。

如果你想要运行多个shell，可以单独在每个终端窗口中运行一个程序，将非交互性质的程序置于后台运行（后面将会介绍），或者了解一下screen程序的使用方法。

2.16.4　后台进程

当你在shell上运行命令时，命令行提示符会暂时消失，命令结束时又重新显示。你可以使用&操作符将进程设置为后台运行，这样提示符会一直显示，你在进程运行过程中可以继续其他操作。例如，如果你要解压缩一个很大的文件（我们将在2.18节介绍），同时又不想干等执行结果，你就可以使用下面的命令：

```
$ gunzip file.gz &
```

Shell会显示后台新进程的PID，然后直接将命令行提示符显示回来，以便你继续进行其他操作。后台进程在你退出系统后仍会一直运行，这比较适用于那些耗时很长的进程。（你可以设置shell让进程在结束时发送通知。）

后台进程的一个缺点是没法和用户交互（甚至会直接从终端获得输入）。它们可以暂停（用fg恢复运行）或终止以便从标准输入获得数据，也可以将数据输出到标准输出和标准错误，这些数据显示在终端屏幕上，有时会和其他正在运行的进程的输出数据混在一起显示，让人难以辨别。

最好的方式是将输出重定向（输入也可以），比如重定向到文件或别的地方（我们已经在2.14节介绍过），这样屏幕上就不会出现杂乱无章的输出数据。

如果后台进程的输出结果杂乱无章，你需要知道如何整理你的终端窗口内容。bash shell和大多数有全屏交互的程序支持CTRL-L命令，它会清空你的屏幕。进程在从标准输入读取数据之前，经常先使用CTRL-R清空当前行，在错误的时间按了错误的键会让情况更糟。如果不小心在bash提示符下按了CTRL-R，你会被切换到一个让人不知所云的反转搜索模式，这时你可以按ESC退出。

2.17　文件模式和权限

Unix系统中的每一个文件都有一组权限值，用来控制你是否能读、写和运行文件。可以使用命令ls -l来查看这些信息。例如：

```
-rw-r--r--❶ 1 juser  somegroup    7041 Mar 26 19:34 endnotes.html
```

❶是文件模式，显示权限及其他附加信息。文件模式由四部分组成，如图2-1。

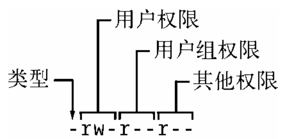


图2-1　文件模式信息

本例中第一个字符-是文件类型，-代表常规文件，常规文件是最常见的一种文件类型，另一种常见类型是目录，用d代表。（3.1节中会介绍其余的文件类型。）

本例中的其余部分是文件权限信息，由三部分组成：用户、用户组和其他。例如，`rw-`是用户权限，后面的`r--`是用户组权限，最后的`r--`是其他权限。

权限信息由四个字符组成：

r　文件可读

w　文件可写

x　文件可执行

-　无

用户权限部分（第一组）是针对文件的拥有者，上例中是juser。用户组权限部分是针对somegroup这个用户组中的所有用户。（命令`groups`可以显示你所在的用户组，详细内容在7.3.5节介绍。）

其他权限部分是针对系统中的所有其他用户，又称为全局权限。

注解　权限信息中代表读、写和执行的这三个部分我们称为权限位，如：读位指的是所有三个代表读的部分。

有些可执行文件的执行位是`s`（setuid）而不是`x`，表示你将以文件拥有者的身份运行该文件，而不是你自己。很多程序使用`s`，如`passwd`命令，因为该命令需要更新/etc/passwd文件，所以必须以文件拥有者（即root用户）的身份运行。

2.17.1　更改文件权限

使用`chmod`命令更改文件权限。例如，对文件file，要为用户组g和其他用户o加上可读权限r，运行以下命令：

```
$ chmod g+r file
$ chmod o+r file
```

也可以使用一行命令：

```
$ chmod go+r file
```

如果要取消权限，则使用`go-r`。

注解　不要将全局权限设置为可写，因为这样任何人都能够修改文件。但是这样会让互联网上的人更改你的文件吗？恐怕不能，除非你的系统有网络安全漏洞。果真是这样的话，文件权限也无能为力。

有时你会看到下面这样的命令，使用数字来代表权限：

```
$ chmod 644 file
```

这个命令会设置所有的权限位，我们称为绝对权限设置。要知道每一个数字（八进制）代表的权限，可以参考该命令的使用手册。请参考下面的表格。

表2-4　绝对权限模式

模　式	详　情	用　途
644	用户：读/写；用户组，其他：读	文件
600	用户：读/写；用户组，其他：无	文件
755	用户：读/写/执行；用户组，其他：读/执行	目录，程序
700	用户：读/写/执行；用户组，其他：无	目录，程序
711	用户：读/写/执行；用户组，其他：执行	目录

和文件一样，目录也有权限。如果你对目录有读的权限，你就可以列出目录中的内容，但是如果要访问目录中的某个文件，你就必须对目录有可执行的权限（使用绝对值设置权限的时候，目录的可执行权限常常会被不小心取消）。

你还可以使用umask命令来为文件设置预定义的默认权限。例如，如果你想让任何人对文件和目录有读的权限，使用umask 022，反之，如果不想让你的文件和目录可读，使用umask 077（在第13章中我们将详细介绍如何在启动文件中使用umask命令）。

2.17.2　符号链接

*符号链接*是指向文件或者目录的文件，相当于文件的别名（类似Windows中的快捷方式）。符号链接为复杂的目录提供了便捷快速的访问方式。

在一个长目录列表里，符号链接如下例所示（请注意文件类型是l）：

```
lrwxrwxrwx 1 ruser users 11 Feb 27 13:52 somedir -> /home/origdir
```

如果你访问somedir，实际访问的是home/origdir目录，符号链接仅仅是指向另一个名字的名字，所以/home/origdir这个目录即使不存在也没有关系。

如果/home/origdir不存在的话，访问somedir的时候系统会报错称somedir不存在。

符号链接不提供其目标路径的详细信息，你只能自己打开这个链接，看看它指向的究竟是文件还是目录。有时候一个符号链接还可以指向另一个符号链接，我们称为*链式符号链接*。

2.17.3　创建符号链接

使用ln -s命令创建符号链接：

```
$ ln -s target linkname
```

linkname参数是符号链接名称，target参数是要指向的目标路径，-s选项表示这是一个符号链接（请见稍后的警告部分）。

运行这个命令之前请反复确认，如果你不小心调换了target和linkname这两个参数的位置，命令变成了：ln -s linkname target，如果linkname这个路径已经存在，一些有趣的事情就会发生。ln会在linkname目录中创建一个名为target的符号链接，如果linkname不是绝对路径，target就会指向它自己。当你使用ln命令遇到问题的时候，请注意检查此类情况。

如果不知道符号链接已经存在的话，就会带来很多麻烦。例如，你有可能无意中会将符号链接文件当作普通文件进行编辑。

警告 创建符号链接的时候，请注意不要忘记-s选项。没有此选项的话，ln命令会创建一个硬链接，为文件创建一个新的名字。新文件拥有老文件的所有状态信息，和符号链接一样，打开这个新文件会直接打开文件内容。除非你掌握了4.5节的内容，否则不要使用符号链接。

符号链接能方便我们管理、组织、共享文件，所以即使有这么多“缺点”，我们还是会用到它。

2.18 归档和压缩文件

了解了文件、权限和相关错误信息之后，让我们来了解一下gzip和tar。

2.18.1 gzip命令

gzip（GNU Zip）命令是Unix上众多标准压缩程序中的一个。GNU Zip生成的压缩文件带有后缀名.gz。解压缩.gz文件使用gunzip file.gz命令，压缩文件使用gzip file命令。

2.18.2 tar命令

gzip命令只压缩单个文件，要压缩和归档多个文件和目录，可以使用tar命令：

```
$ tar cvf archive.tar file1 file2 ...
```

tar命令生成的文件带有后缀名.tar，<archive>.tar是生成的归档文件名，file1、file2等是要归档的文件和目录列表。选项c代表创建文件。选项r和f的作用则更加具体。

选项v用来显示详细的命令执行信息（比如正在归档的文件和目录名），再加一个v选项可以显示文件大小和权限等信息。如果你不想看到这些信息，可以不用加v选项。

选项f代表文件，后面需要指定一个归档文件名（如<archive>.tar）。如果不指定归档文件名，则归档到磁带设备，如果文件名为-,则是归档到标准输入或者输出。

解压缩tar文件

使用tar命令解压缩.tar文件：

```
$ tar xvf archive.tar
```

选项x代表解压模式。你还可以只解压归档文件中的某几个文件，只需要在命令后面加上这些文件的文件名即可。

注解　tar命令解压后并不会删除归档文件。

内容预览表模式

在解压一个归档文件之前，通常建议使用选项t来查看归档文件中的内容，t代表内容预览表模式，它会显示归档的文件列表，并且验证归档信息的完整性。如果你不做检查直接解压归档文件，有时会解压出一些很难清理的垃圾内容。

你需要检查压缩包中的文件是否在同一目录下，你可以创建一个临时目录，在其中试着解压一下看看。

解压缩时，你可以使用选项p来保留被归档文件的权限信息。当你使用超级用户运行解压命令时，选项p默认开启。如果你在执行过程中遇到这样那样的问题，请确保你等到tar命令执行完毕并显示提示符。tar命令每次都处理整个归档文件，无论你是解压整个文件或者只是文件中的某部分，所以命令执行过程中请不要中断，因为有些操作是在文件检查之后才开始的，比如权限设置。

2.18.3　压缩归档文件（.tar.gz）

许多初学者对被压缩后的归档文件（后缀为.tar.gz）比较费解。我们可以按照从右到左的顺序来解压和打开此类文件。例如，使用以下命令首先解压缩，然后校验及释放归档文件包：

```
$ gunzip file.tar.gz
$ tar xvf file.tar
```

如果需要归档并压缩，则按照相反顺序，先运行tar命令归档，然后运行gzip命令压缩。你可能会逐渐觉得这两个步骤很麻烦，下面我们介绍一些更简便的方法。

2.18.4　zcat命令

上面的命令缺点是执行效率不高，并且会占用很多硬盘空间。管道命令是一个更好的选择，例如：

```
$ zcat file.tar.gz | tar xvf -
```

zcat命令等同于gunzip -dc命令。选项d代表解压缩，选项c代表将运行结果输出到标准输出（本例中是输出到tar命令）。

tar命令很常用，它的Linux版本有这样几个选项值得注意：选项z对归档文件自动运行gzip，对创建归档包和释放归档包均适用。例如使用以下命令来验证压缩文件：

```
$ tar ztvf file.tar.gz
```

然而，在走捷径以前，最好还是先掌握基本方法。

注解 .tgz文件和.tar.gz文件没有区别，后缀.tgz主要是针对MS-DOS的FAT文件系统。

2.18.5 其他的压缩命令

Unix中的另一个压缩命令是bzip2，生成后缀名为.bz2的文件。该命令执行效率比gzip稍慢，主要用来压缩文本文件，因而在压缩源代码文件的时候比较常用。相应的解压缩命令是bunzip2，可用选项和gzip几乎相同，其中选项j是针对tar命令的压缩和解压缩。

此外xz是另外一个渐受欢迎的压缩命令，对应的解压缩命令是unxz，可用选项和gzip也十分类似。

Linux上的zip和unzip与Windows上的.zip文件格式大部分是兼容的，包括.zip和.exe自解压文件。有一个很古老的Unix命令compress支持.Z格式，gunzip命令能够解压缩.Z文件，但是gzip不支持此格式文件的创建。

2.19 Linux 目录结构基础

我们前面介绍了文件、目录和帮助手册，现在来看看系统文件。Linux目录结构的详解可以参考文件系统标准结构（FHS，http://www.pathname.com/fhs/）。图2-2为我们展示了Linux的基本目录结构，包括目录/、/usr和/var下的子目录。请注意/usr下的子目录有些和/下的子目录一样。

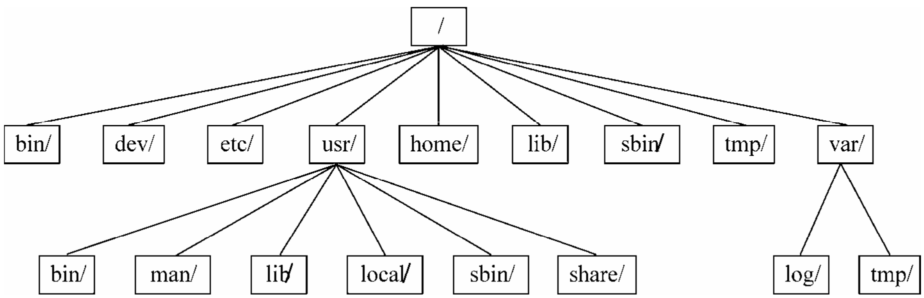


图2-2 Linux目录结构

下面这些目录需要重点介绍。

- /bin目录中存放的是可执行文件，包括大部分基础的Unix命令（如ls和cp）。该目录中的大部分是由C编译器创建的二进制文件，还有一些现代系统的shell脚本文件。
- /dev目录中是设备文件，将在第3章详细介绍。
- /etc目录（读作EHT-see）存放重要的系统配置文件，如用户密码文件、启动文件、设备、网络和其他配置文件。许多都是硬件系统的配置文件。例如，/etc/X11目录中是显示卡和视窗系统的配置文件。
- /home目录中是用户的个人目录。大多数Unix系统都遵循这个规范。
- /lib目录中是供可执行程序使用的各种代码库。代码库分为两种：静态库和共享库。/lib目录中一般只有共享库。其他代码库目录，如：/usr/lib中会有静态库和动态库，以及其他的辅助文件（将在第15章详细介绍）。
- /proc目录中通过一个可浏览的目录与文件接口来存放系统相关信息，比如当前运行的进程和内核的信息。Linux上这个目录的一大部分子目录结构相比其他Unix系统要特别一些，但其他Unix系统大多也有类似的特性。/proc目录中包含了当前正在运行的进程的信息以及一些内核参数。
- /sys目录类似/proc目录，里面是设备和系统的信息（将在第3章介绍）。
- /sbin目录中是可执行的系统文件，这些可执行文件用来管理系统。普通用户一般不需要使用，许多命令只能由root用户运行。
- /tmp目录存放无关紧要的临时文件。所有用户对该目录都有读和写的权限，不过可能对别人的文件没有权限。许多程序会使用这个目录作为保存数据的地方，但如果数据很重要的话请不要存放在/tmp目录中，因为很多系统会在启动时清空/tmp目录，甚至是经常性地清理这个目录里的旧文件。也注意不要让/tmp目录里的垃圾文件占用太多的硬盘空间。
- /usr目录虽然读作user，但里面并没有用户文件，而是存放着许多Linux系统文件。/usr目录中的很多目录名和root目录上的相同（如/usr/bin和/usr/lib），里面存放的文件类型也相同。（为了让root文件系统占用尽可能少的空间，许多系统文件并没有存放在系统root目录下。）
- /var目录是程序存放运行时信息的地方，如系统日志、用户信息、缓存和其他信息。（你可能会注意到这里有一个子目录/var/tmp，和/tmp不同的是，系统不会在启动时清空它。）

2.19.1　root目录下的其他目录

root目录下还有以下这些子目录。

- /boot目录存放内核加载文件。这些文件中存放Linux在第一次启动时的信息，之后的信息并不保存在这里。详见第5章。
- /media目录是加载可移除设备的地方，比如可移动硬盘。
- /opt目录一般存放第三方软件，许多系统并没有这个目录。

2.19.2　/usr目录

/usr目录中内容比它的名字多得多，你看一看/usr/bin和/usr/lib的内容就会知道，/usr目录存放那些运行在用户空间中的进程和数据。除了/usr/bin、/usr/sbin1和/usr/lib外，还包括以下内容。

- /include目录存放C编译器需要使用的头文件。
- /info目录存放GNU帮助手册（参考2.13节）。
- /local目录是管理员安装软件的地方，它的结构和/以及/usr类似。
- /man存放用户手册。
- /share目录存放Unix系统间的共享文件。过去这个目录通常在网络中被共享，现在使用得越来越少，因为硬盘空间不再是一个大问题，且维护/share目录也让人头疼。/share目录中经常包括/man、/info和其他子目录。

2.19.3　内核位置

Linux系统的内核通常在/vmlinuz或者/boot/vmlinuz中。系统启动时，引导装载程序将这个文件加载到内存并运行（我们将在第5章详细介绍）。

引导装载程序执行完毕后，系统就不再需要内核文件了。不过，系统在运行过程中会根据需要加载和卸载许多模块，我们称之为*可加载内核模块*，它们在/lib/modules目录下可以找到。

2.20　以超级用户的身份运行命令

继续新内容之前，你需要了解如何以超级用户的身份运行命令。你可能已经知道使用su命令然后输入root用户密码就可以启动root命令行，不过这个方法有以下几个缺点：

- 对更改系统的命令没有记录信息；
- 对运行上述命令的用户身份没有记录信息；
- 无法访问普通Shell环境；
- 必须输入root密码。

2.20.1　sudo命令

大部分Linux系统中，管理员可以使用自己的用户账号登录，然后使用sudo来以root用户身份执行命令。例如，在第7章中，我们会介绍如何使用vipw命令编辑/etc/passwd文件。例如：

```
$ sudo vipw
```

执行sudo命令时，它会使用local2中的系统日志服务将操作写入日志。我们将在第7章详细介绍。

2.20.2　/etc/sudoers

系统当然不会允许任何用户都能够以超级用户的身份运行命令，你需要在/etc/sudoers文件中

加入指定的用户。sudo命令有很多选项，使用起来比较复杂。例如，下面的设置允许用户user1和user2不用输入密码即可以超级用户身份运行命令：

```
User_Alias ADMINS = user1, user2

ADMINS  ALL = NOPASSWD: ALL

root    ALL=(ALL) ALL
```

第一行为user1和user2指定一个ADMINS别名，第二行赋予它们权限。ALL = NOPASSWD: ALL表示有ADMINS别名的用户可以运行sudo命令。该行中第二个ALL代表允许执行任何命令，第一个ALL表示允许在任何主机运行命令（如果你有多个主机，你可以针对某个主机或者某一组主机设置，这个我们在这里不详细介绍）。

root ALL=(ALL) ALL表示root用户能够在任何主机上执行任何命令。(ALL)表示root用户可以以任何用户的身份运行命令。你可以通过以下方式将(ALL)权限赋予有ADMINS别名的用户，将(ALL)加入到/etc/sudoers行，如❶所示：

```
ADMINS ALL = (ALL)❶ NOPASSWD: ALL
```

注解　可以使用visudo命令编辑/etc/sudoers文件，该命令在保存文件时会做语法检查。

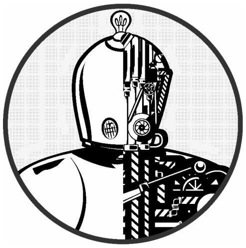
sudo命令我们现在就介绍到这里，详细的使用方法请参考sudoers(5)和sudo(8)帮助手册。（用户切换的详细内容我们将在第7章介绍。）

2.21　前瞻

目前为止你对以下这些命令已经有所了解：运行程序、重定向输出、文件和目录操作、查看进程、查看帮助手册以及用户空间。你应该也学会了以超级用户身份运行命令。关于用户空间组件和内核的详细内容，你可能还不甚了解，但掌握了文件和进程的基础知识后，这些也不再是难事。在下面的章节中，我们就来介绍如何使用这些命令来操作内核和用户空间。

第3章

设备管理



本章介绍与Linux系统内核提供的设备相关的基础设施。纵观Linux发展史，内核向用户呈现设备的方式发生了很大变化。我们将从传统的设备文件系统开始，介绍内核如何通过sysfs来提供设备配置信息。我们的目标是能够通过在系统上收集设备信息来了解一些基本操作。后面的章节将进一步介绍一些具体设备的管理。

理解内核怎样在用户空间呈现新设备很关键。udev系统让用户空间进程能够自动配置和使用新设备。我们将介绍内核如何通过udev向用户空间进程发送消息，以及进程如何处理这些消息。

3.1 设备文件

在Unix系统中操纵大多数设备都很容易，因为很多I/O接口都是以文件的形式由内核呈现给用户的。这些设备文件有时又叫作*设备节点*。开发人员可以像操作文件一样来操作设备，一些Unix标准命令（如cat）也可以访问设备，所以不仅仅开发人员，普通用户也能够访问设备。然而对文件接口所能执行的操作是有限制的，所以并不是所有设备或设备功能都能够通过标准文件I/O方式来访问。

Linux处理设备文件的方式和Unix一样。设备文件存放在/dev目录中，可以使用`ls /dev`命令来查看。

我们从下面这个命令开始：

```
$ echo blah blah > /dev/null
```

这个命令将执行结果从标准输出重定向到一个文件，这个文件是/dev/null，它是一个设备，内核决定如何处理设备的数据写入。对/dev/null来说，内核直接忽略输入数据。

你可以使用ls -l来查看设备及其权限。

例3-1　设备文件

```
$ ls -l
brw-rw----    1 root disk 8, 1 Sep  6 08:37 sda1
crw-rw-rw-    1 root root 1, 3 Sep  6 08:37 null
prw-r--r--    1 root root    0 Mar  3 19:17 fdata
srw-rw-rw-    1 root root    0 Dec 18 07:43 log
```

请注意，上面每一行的第一个字符（代表文件模式）：字符b（block）、c（character）、p（pipe）和s（socket）代表设备文件。下面是详细介绍。

块设备

程序从块设备中按固定的块大小读取数据。前面的例子中，sda1是一个*磁盘设备*，它是块设备的一种。我们能够轻松地将磁盘划分成数据区块。因为磁盘的容量是固定的，索引起来也很方便，所以进程能够通过内核访问磁盘上的任意区块。

字符设备

*字符设备*处理流数据。你只能对字符设备读取和写入字符数据，如前面例子中的/dev/null。字符设备没有固定容量，当你对字符设备进行读写时，内核对相应的设备进行读写操作。字符设备的一个例子是打印机，值得注意的是，内核在流数据送达设备和进程后不会备份和再次验证。

管道设备

*命名管道*设备和字符设备类似，不同的是输入输出端不是内核驱动程序，而是另外一个进程。

套接字设备

*套接字*设备是跨进程通信经常用到的特殊接口。它们经常会存放于/dev目录之外。套接字文件代表Unix域套接字，我们将在第10章详细介绍。

在例3-1的第1、2行中，日期前的两个数字代表*主要*和*次要*设备号，它们是内核用来识别设备的数字。相同类型的设备一般有相同的主设备号，比如sda3和sdb1（它们都是磁盘分区）。

注解　*并不是所有的设备都有对应的设备文件，因为块设备和字符设备并不是适合所有场合的。例如，网络接口没有设备文件，虽然其理论上可以使用字符设备来代表，但是实现起来实在很困难，所以内核采用了其他的I/O接口。*

3.2　sysfs 设备路径

传统的Unix /dev目录为用户进程与使用内核支持的设备进行引用与交互提供了便利，但是它过于简单。/dev目录中的文件名包含有关设备的一些信息，但不是很详尽。另一个问题是内核根据其找到设备的顺序为设备文件命名，所以系统每次重新启动后，设备文件名有可能不同。

Linux内核通过一个文件和目录系统提供sysfs界面，旨在基于硬件属性统一显示设备的相关

信息。设备以/sys/devices为root路径。例如，/dev/sda代表的SATA硬盘在sysfs中的路径可能是：

```
/sys/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda
```

你可以看到，这个路径比文件名/dev/sda长很多，后者也是一个目录。但你实际上不能对比这两个路径，因为它们的作用不一样。/dev目录中的文件是供用户进程使用设备的，而/sys/devices中的文件是用来查看设备信息和管理设备用的。如果你打开上述设备路径，就能够看到类似下面的内容：


```
alignment_offset  discard_alignment  holders   removable  size       uevent
bdi               events             inflight  ro         slaves
capability        events_async       power     sda1       stat
dev               events_poll_msecs  queue     sda2       subsystem
device            ext_range          range     sda5       trace
```

这些文件和子目录一般都是供程序而不是用户访问的，但你可以通过诸如/dev文件这样的例子来了解它们包含和代表的内容。运行命令`cat dev`会显示数字8:0，这刚好是/dev/sda设备的主要和次要编号。

/sys目录下有几个快捷方式。例如，/sys/block目录中包含系统中的所有块设备文件，不过它们都是符号链接。运行命令`ls -l /sys/block`可以显示指向sysfs的实际路径。

在/dev目录中查看设备文件的sysfs路径不太方便，可以使用udevadm命令来查看路径和其他属性：

```
$ udevadm info --query=all --name=/dev/sda
```

注解 udevadm命令在/sbin目录下，如果你的路径中没有，可以将该目录加到你的路径中。

udevadm和udev系统将在3.5节详细介绍。

3.3 dd 命令和设备

dd命令对于块设备和字符设备非常有用，它的主要功能是从输入文件和输入流读取数据然后写入输出文件和输出流，在此过程中可能涉及到编码转换。

dd命令复制固定大小的数据块，例如下面的代码显示如何借助字符设备使用dd命令：

```
$ dd if=/dev/zero of=new_file bs=1024 count=1
```

dd命令的格式选项和大多数其他Unix命令不同，它沿袭了从前的IBM Job Control Language（JCL）的风格。它使用等号=而不是减号-来设定选项和参数值。上面的例子是从/dev/zero复制一个大小为1024字节的数据块到文件new_file。

以下是dd命令的一些重要选项。

- if=file：代表输入文件，默认是标准输入。
- of=file：代表输出文件，默认是标准输出。
- bs=size：代表数据块大小。dd命令一次读取或者写入数据的大小。对于海量数据，你可以在数字后设置b和k来分别代表512字节和1024字节。如：bs=1k和bs=1024一样。
- ibs=size，obs=size：代表输入和输出块大小。如果输入输出块大小相同，你可以使用bs选项，如果不相同的话，可以使用ibs和obs分别指定。
- count=num：代表复制块的总数。在处理大文件或者无限数据流（/dev/zero）的时候，你可能会需要在某个地方停止dd复制，不然的话将会消耗大量硬盘空间和CPU时间。这时你可以使用count和skip选项从大文件或设备中复制一小部分数据。
- skip=num：代表跳过前面的num个块，不将它们复制到输出。

警告 dd命令功能非常强大，你需要先对其充分了解再使用，否则稍一疏忽就会损坏文件和设备上的数据。dd命令通常用来将输出数据写入到新文件。

3.4 设备名总结

有时候查找设备的名称不是很方便（比如在为硬盘分区的时候），下面我们介绍一些简便的方法。

- 使用udevadm命令来查询udevd（见3.5节）。
- 在/sys目录下查找设备。
- 从dmesg（它显示最新的内核消息，见7.2节）命令的输出或者内核系统日志中查获设备名。这些地方通常会有系统设备的描述信息。
- 对系统已经找到的硬盘设备，可以使用mount命令查看结果。
- 运行cat /proc/devices命令，以查看系统为之配备了驱动程序的块设备和字符设备。输出结果中的每一行包含设备的主要编号和名称（见3.1节）。你可以根据主要编号到/dev目录中查找对应的块设备和字符设备文件。

这些方法中只有第一个方法比较可靠，但是它需要udev。如果你的系统中没有udev的话，你可以尝试其他几个方法，尽管有时候内核并没有一个设备文件来对应你要找的设备。

下面我们列出一些最常见的Linux设备及其命名规范。

3.4.1 硬盘：/dev/sd*

目前Linux系统中的硬盘设备大部分都以sd为前缀来命名，如/dev/sda，/dev/sdb等。这些设备代表整块硬盘，内核使用单独的设备文件名来代表硬盘上的分区，如/dev/sda1、/dev/sda2。

这里需要进一步解释一下命名规范。sd代表SCSI disk。小型计算机系统接口（Small Computer System Inteface，以下简称SCSI）最初是作为设备之间通信的硬件协议标准而开发的，虽然现在

的计算机并没有使用传统的SCSI硬件，但是SCSI协议的运用却非常广泛。例如，USB存储设备就使用SCSI协议进行通信。SATA硬盘的情况相对复杂一些，但是Linux内核仍然在某些场合使用SCSI命令和它们通信。

我们可以使用sysfs系统提供的命令来查看系统中的SCSI设备。最常用的命令之一是lsscsi。运行结果如下例所示：


```
$ lsscsi
[0:0:0:0]❶    disk❷  ATA      WDC WD3200AAJS-2 01.0 /dev/sda❸
[1:0:0:0]     cd/dvd Slimtype DVD A   DS8A5SH  XA15 /dev/sr0
[2:0:0:0]     disk   FLASH    Drive  UT_USB20  0.00 /dev/sdb
```

上例中，❶字段是指设备在系统中的地址，❷字段是设备的描述信息，❸字段则是设备文件的路径。其余的是设备提供商的相关信息。

Linux按照设备驱动程序检测到设备的顺序来分配设备文件。在前面的例子中，内核先检测到磁盘，然后是cd/dvd，最后是闪存盘。

悲剧的是，这种方式在重新配置硬件时会导致一些问题。比如说你的系统有三块硬盘：/dev/sda、/dev/sdb和/dev/sdc，如果/dev/sdb损坏了，你必须将其移除才能使系统正常工作，然而/dev/sdc已经不存在了，之前的/dev/sdc现在成了/dev/sdb。如果你在fstab文件（见4.2.8节）中引用了/dev/sdc，你就必须更新此文件。为了解决这个问题，大部分现代的Linux系统使用通用唯一标识符（Universally Unique Identifier，以下简称UUID，见4.2.4节）来访问设备。

这里提到的内容不涉及硬盘和其他存储设备的使用细节，相关内容我们将在第4章介绍。在本章稍后我们会介绍SCSI如何支持Linux内核的运行。

3.4.2　CD和DVD：/dev/sr*

Linux系统能够将大多数光学存储设备识别为SCSI设备，如/dev/sr0、/dev/sr1等。但是如果光驱使用的是老接口的话，可能会被识别为PATA设备。/dev/sr*设备是只读的，它们只用于从光盘上读取数据。可读写光盘驱动用/dev/sg0这样的设备文件表示，g代表“generic”。

3.4.3　PATA硬盘：/dev/hd*

老版本的Linux内核常用设备文件/dev/hda、/dev/hdb、/dev/hdc和/dev/hdd来代表老的块设备。这是基于主从设备接口0和1的固定设置方式。SATA设备有时候也会被这样识别，这表示SATA设备在兼容模式中运行，会造成性能损失。你可以检查你的BIOS设置，看看能否将SATA控制器切换到它原有的模式。

3.4.4　终端设备/dev/tty/*、/dev/pts/*和/dev/tty

终端设备负责在用户进程和输入输出设备之间传送字符，通常是在终端显示屏上显示文字。终端设备接口由来已久，一直可以追溯到手动打字机时代。

伪终端设备模拟终端设备的功能，由内核为程序提供I/O接口，而不是真实的I/O设备，shell窗口就是伪终端。

常见的两个终端设备是/dev/tty1（第一虚拟控制台）和/dev/pts/0（第一虚拟终端），/dev/pts目录中有一个专门的文件系统。

/dev/tty代表当前进程正在使用的终端设备，虽然不是每个进程都连接到一个终端设备。

显示模式和虚拟控制台

Linux系统有两种显示模式：文本模式和X Windows系统服务器（即图形模式，通常是通过显示管理器）。通常系统是在文本模式下启动，但是很多Linux发行版通过内核参数和内置图形显示机制（如plymouth）将文本模式完全屏蔽起来，这样系统从始至终是在图形模式下启动。

Linux系统支持虚拟控制台来实现多个终端的显示，虚拟控制台可以在文本模式和图形模式下运行。在文本模式下，你可以使用ALT-Function在控制台之间进行切换，例如ALT-F1切换到/dev/tty1，ALT-F2切换到/dev/tty2等等。这些控制台通常会被getty进程占用以显示登录提示符，详见7.4节。

X server在图形模式下使用的虚拟控制台稍微有些不同，它不是从init配置中获得虚拟控制台，而是由X server来控制一个空闲的虚拟控制台，除非另外指定。例如，如果tty1和tty2上运行着getty进程，X server就会使用tty3。此外，X server将虚拟控制台设置为图形模式后，通常需要按CTRL-ALT-Function而不是ALT-Function来切换到其他虚拟控制台。

如果你想在系统启动后使用文本模式，可以按CTRL-ALT-F1。按ALT-F2、ALT-F3等返回X11会话。

如果在切换控制台的时候遇到问题，你可以尝试chvt命令强制系统切换工作台。例如：使用root运行以下命令切换到tty1：

```
# chvt 1
```

3.4.5　串行端口：/dev/ttyS*

老式的RS-232和串行端口是特殊的终端设备，串行端口设备在命令行上运用不太广，原因是需要处理诸如波特律和流控制等参数的设置。

Windows上的COM1端口在Linux中表示为/dev/ttyS0，COM2表示为/dev/ttyS1，以此类推。可插拔USB串行适配器在USB和ACM模式下分别表示为：/dev/ttyUSB0、/dev/ttyACM0、/dev/ttyUSB1、/dev/ttyACM1等。

3.4.6　并行端口：/dev/lp0 和/dev/lp1

单向并行端口设备，目前被USB广泛取代的一种接口类型，表示为：/dev/lp0和/dev/lp1，分别代表Windows中的LPT1:和LPT2:。你可以使用cat命令将整个文件（比如说要打印的文件）发送到并行端口，执行完毕后你可能需要向打印机发送另外的指令（`form feed`或`reset`）。像CUPS

这样的打印服务相比打印机来说提供了更好的用户交互体验。双向并行端口表示为：/dev/parport0和/dev/parport1。

3.4.7 音频设备：/dev/snd/*、/dev/dsp、/dev/audio和其他

Linux系统有两组音频设备，分别是高级Linux声音架构（Advanced Linux Sound Architecture，以下简称ALSA）和开放声音系统（Open Sound System，以下简称OSS）。ALSA在/dev/snd目录下，要直接使用不太容易。如果Linux系统中加载了OSS内核支持，则ALSA可以向后兼容OSS设备。

OSS dsp和音频设备支持一些基本的操作。例如，可以将WAV文件发送给/dev/dsp来播放。然而如果频率不匹配的话，硬件有可能无法正常工作。并且在大多数系统中，音频设备在你登录时通常处于忙状态。

> **注解** Linux的音频处理非常复杂，因为涉及很多层细节。我们刚刚介绍的是内核级设备，通常在用户空间中还有puls-audio这样的服务来负责处理不同来源和声音设备的音频处理。

3.4.8 创建设备文件

在现代Linux系统中，你不需要创建自己的设备文件，这项工作是由devtmpfs和udev（见3.5节）来完成。不过了解一下这个过程总是有益的，以备不时之需。

mknod命令用来创建设备。你必须知道设备名以及主要和次要编号。例如，可以使用以下命令创建设备/dev/sda1：

```
# mknod /dev/sda1 b 8 2
```

参数b 8 2分别代表块设备、主要编号8和次要编号2。字符设备使用c，命名管道使用p（主要和次要编号可忽略）。

用mknod命令来创建临时的命名管道很方便，也可以用于在系统恢复的时候创建丢失的设备文件。

在老版本的Unix和Linux系统中，维护/dev目录不是一件容易的事情。内核每次更新和增加新的驱动程序，能支持的设备就更多，同时也意味着一些新的主要和次要编号被指定给设备文件。为了方便维护，系统使用/dev目录下的MAKEDEV程序来创建设备组。在系统升级的时候你可以看看有没有新版本的MAKEDEV，如果有的话可以运行它来创建新设备。

这样的静态管理系统非常不好用，所以出现了一些新的选择。首先是devfs，它是/dev在内核空间的一个实现版本，包含内核支持的所有设备。但是它的种种局限使得人们又开发了udev和devtmpfs。

3.5　udev

我们已经介绍过，内核中的一些豪无必要的复杂功能会降低系统的稳定性。设备文件管理就是一个很好的例子：如果你可以在用户空间内创建设备文件的话，就不需要在内核空间做。Linux系统内核在检测到新设备的时候（如发现一个USB存储器），会向用户空间进程发送消息（称为udevd）。用户空间进程会在另一端验证新设备的属性，创建设备文件，执行初始化。

理论上是如此，但实际上这个方法有一些问题：系统启动前期即需要设备文件，所以udevd需要在其之前启动。udevd不能依赖于任何设备来创建设备文件，它必须尽快启动以免拖延整个系统。

3.5.1　devtmpfs

devtmpfs文件系统正是为了解决上述问题而开发的（详见4.2节）。它类似老的devfs系统，但是更简单。内核根据需要创建设备文件，并且在新设备可用时通知udevd。udevd在收到通知后并不创建设备文件，而是进行设备初始化以及发送消息通知。此外还在/dev目录中为设备创建符号链接文件。你可以在/dev/disk/by-id目录中找到一些实例，其中每个硬盘对应一个或者多个文件。

例如下面的例子：

```
lrwxrwxrwx 1 root root  9 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD- WMAV2FU80671 -> ../../sda
lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD- WMAV2FU80671-part1 ->
  ../../sda1
lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD- WMAV2FU80671-part2 ->
  ../../sda2
lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD- WMAV2FU80671-part5 ->
  ../../sda5
```

udevd使用接口类型名称、厂商、型号、序列号以及分区（如果有的话）的组合来命名符号链接。

下一节介绍udevd是怎样创建符号链接文件的，不过你现在并不需要马上了解。实际上，如果你是第一次接触Linux设备管理，你可以直接跳到下一章去了解如何使用硬盘。

3.5.2　udevd的操作和配置

udevd守护进程是这样工作的。

(1) 内核通过一个内部网络链接向udevd发送一个通知事件，称作uevent。

(2) udevd加载uevent中的所有属性信息。

(3) udevd通过规则解析来决定执行哪些操作和增加哪些属性信息。

呼入uevent是udevd从内核接收到的消息，如下面代码所示：

```
ACTION=change
DEVNAME=sde
```

```
DEVPATH=/devices/pci0000:00/0000:00:1a.0/usb1/1-1/1-1.2/1-1.2:1.0/host4/
  target4:0:0/4:0:0:3/block/sde
DEVTYPE=disk
DISK_MEDIA_CHANGE=1
MAJOR=8
MINOR=64
SEQNUM=2752
SUBSYSTEM=block
UDEV_LOG=3
```


你能够看到上例对设备做了一处修改。接收到uevent以后，udevd获得了sysfs的设备路径和一些属性信息，现在可以执行规则解析了。

规则文件位于/lib/udev/rules.d和/etc/udev/rules.d目录中。默认规则在/lib目录中，会被/etc中的规则覆盖。有关规则的详细内容非常多，你可以参考udev(7)帮助手册。现在让我们看一下3.5.1一节中/dev/sda一例中的符号链接。这些链接是在/lib/udev/rules.d/60-persistent-storage.rules中定义的。你能够在其中找到如下内容：

```
# ATA devices using the "scsi" subsystem
KERNEL=="sd*[!0-9]|sr*", ENV{ID_SERIAL}!="?*", SUBSYSTEMS=="scsi",ATTRS{vendor}=="ATA",
  IMPORT{program}="ata_id --export $tempnode"
# ATA/ATAPI devices (SPC-3 or later) using the "scsi" subsystem
KERNEL=="sd*[!0-9]|sr*", ENV{ID_SERIAL}!="?*", SUBSYSTEMS=="scsi",ATTRS{type}=="5",
  ATTRS{scsi_level}=="[6-9]*", IMPORT{program}="ata_id --export $tempnode"
```

这些规则和内核SCSI子系统呈现的ATA硬盘相匹配（参见3.6节）。你可以看到udevd尝试匹配以sd或者sr开头，但是不包含数字的设备名（通过表达式：`KERNEL=="sd\*[!0-9]|sr\*"`）、匹配子系统（`SUBSYSTEMS=="scsi"`）和其他一些属性。如果上述所有条件都满足，则进行下一步：

```
IMPORT{program}="ata_id --export $tempnode"
```

这不是一个条件，而是一个指令，它从`/lib/udev/ata_id`命令导入变量。如果你有匹配的设备，可以试着执行以下命令行：

```
$ sudo /lib/udev/ata_id --export /dev/sda
ID_ATA=1
ID_TYPE=disk
ID_BUS=ata
ID_MODEL=WDC_WD3200AAJS-22L7A0
ID_MODEL_ENC=WDC\x20WD3200AAJS22L7A0\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20
  \x20\x20\x20\x20\x20\x20\x20\x20\x20
ID_REVISION=01.03E10
ID_SERIAL=WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671
--snip--
```

上面所有变量名都被设置了相应的值，`ENV{ID_TYPE}`的值对后面的规则都为`disk`。

`ID_SERIAL`需要特别注意一下，在每一个规则中都有这个条件行：

```
ENV{ID_SERIAL}!="?*"
```

意思是如果ID_SERIAL变量没有被设置，条件语句返回true，反之如果变量被设置，则为false，当前规则返回false，udevd继续解析下一规则。

这是什么意思呢？这两条规则的目的是找出硬盘设备的序列号，如果ENV{ID_SERIAL}被设置，udevd就能够解析下面的规则：

```
KERNEL=="sd*|sr*|cciss*", ENV{DEVTYPE}=="disk", ENV{ID_SERIAL}=="?*",
  SYMLINK+="disk/by-id/$env{ID_BUS}-$env{ID_SERIAL}"
```

你可以看到这个规则要求ENV{ID_SERIAL}被赋值，它有如下指令：

```
SYMLINK+="disk/by-id/$env{ID_BUS}-$env{ID_SERIAL}"
```

执行这个指令时，udevd为新加入的设备创建一个符号链接。现在我们可以知道设备符号链接的来由了。

你也许会问如何在指令中判断条件表达式，条件表达式使用==或!=，而指令使用=、+=或者:=。

3.5.3　udevadm

udevadm程序是udevd的管理工具，你可以使用它来重新加载udevd规则，触发消息。它功能强大之处在于搜寻和浏览系统设备以及监控udevd从内核接收的消息。使用udevadm需要掌握一些命令行语法。

我们首先来看看如何检验系统设备。回顾一下3.5.2节中的例子，我们使用以下命令来查看设备（如/dev/sda）的udev属性和规则：

```
$ udevadm info --query=all --name=/dev/sda
```

运行结果如下：

```
P:/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda
N: sda
S: disk/by-id/ata-WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671
S: disk/by-id/scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671
S: disk/by-id/wwn-0x50014ee057faef84
S: disk/by-path/pci-0000:00:1f.2- scsi-0:0:0:0
E: DEVLINKS=/dev/disk/by-id/ata-WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671 /dev/disk/by-id/scsi
  -SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671 /dev/disk/by-id/wwn-0x50014ee057faef84 /dev/disk/by
  -path/pci-0000:00:1f.2-scsi-0:0:0:0
E: DEVNAME=/dev/sda
E: DEVPATH=/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda
E: DEVTYPE=disk
E: ID_ATA=1
E: ID_ATA_DOWNLOAD_MICROCODE=1
E: ID_ATA_FEATURE_SET_AAM=1
--snip--
```

其中每一行的前缀代表设备的属性值，如P:代表sysfs设备路径，N:代表设备节点（/dev下的设备文件名），S:代表指向设备节点的符号链接，由udevd在/dev目录中根据其规则生成，E:代表从udevd规则中获得的额外信息。（另外还有很多其他的信息，你可以自己运行命令看一看。）

3.5.4　设备监控

在udevadm中监控uevent可以使用monitor命令：

```
$ udevadm monitor
```

例如，如果你插入一个闪存盘，该命令执行结果如下：

```
KERNEL[658299.569485] add      /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2 (usb)
KERNEL[658299.569667] add      /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0 (usb)
KERNEL[658299.570614] add      /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0/host15
  (scsi)
KERNEL[658299.570645] add      /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0/
  host15/scsi_host/host15 (scsi_host)
UDEV [658299.622579] add       /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2 (usb)
UDEV [658299.623014] add       /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0 (usb)
UDEV [658299.623673] add       /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0/host15
  (scsi)
UDEV [658299.623690] add       /devices/pci0000:00/0000:00:1d.0/usb2/2- 1/2-1.2/2-1.2:1.0/
  host15/scsi_host/host15 (scsi_host)
--snip--
```

上面结果中，每个消息对应有两行信息，因为命令默认显示从内核接收的呼入消息（用KERNEL标记）和udevd在处理该消息时发送给其他程序的消息。如果只想看内核发送的消息，可以使用--kernel选项，只看udev发送的消息可以用--udev。查看呼入消息的所有属性（见3.5.2节）可以使用--property选项。

你还可以使用子系统来过滤消息。例如，如果只想看和SCSI有关的内核消息，可以使用下面的命令：

```
$ udevadm monitor --kernel --subsystem-match=scsi
```

udevadm的更多内容可以参考udevadm(8)使用手册。

关于udev还有很多内容，比如处理中间通信的D-Bus系统有一个守护进程叫作udisks-daemon，它通过监听udevd的呼出消息来自动通知桌面应用系统发现了新的硬盘。

3.6　详解 SCSI 和 Linux 内核

本节我们将介绍Linux内核对SCSI的支持，借此机会了解一下Linux内核的架构。本节的内容偏理论，如果你想先了解如何使用硬盘，可以直接跳到第4章。

首先我们介绍一下SCSI的背景知识，传统的SCSI硬件设置是通过SCSI总线链接设备到主机适配器，如图3-1所示。主机适配器和设备都有一个SCSI ID，每个总线有8到16个ID（不同版本数量不同）。SCSI目标指的是设备及其SCSI ID。

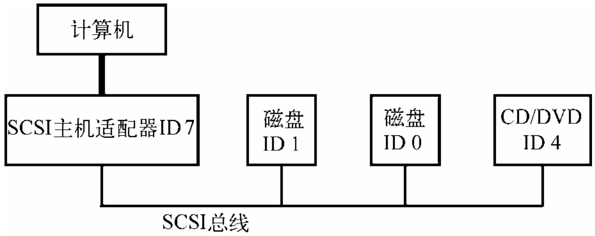


图3-1　有主机适配器和设备的SCSI总线

计算机并不和设备链直接连接，所以必须通过主机适配器和设备通信。主机适配器通过SCSI命令集与设备进行一对一通信，设备向其发送响应消息。

更新版本的SCSI如Serial Attached SCSI（SAS）的性能更出色，不过大部分计算机中并没有真正意义的SCSI设备。更多的是那些使用SCSI命令的USB存储设备。支持ATAPI的设备（如CD/DVD-ROM）也使用某个版本的SCSI命令集。

SATA硬盘在系统中通常由一个处于libata层（见3.6.2节）的转换机制呈现为SCSI设备。一些SATA控制器（特别是高性能RAID控制器）使用硬件来实现这个转换。

让我们用下面这个例子将上述内容整合起来：

```
$ lsscsi
[0:0:0:0]    disk     ATA       WDC  WD3200AAJS-2 01.0  /dev/sda
[1:0:0:0]    cd/dvd   Slimtype  DVD A DS8A5SH     XA15  /dev/sr0
[2:0:0:0]    disk     USB2.0    CardReader CF     0100  /dev/sdb
[2:0:0:1]    disk     USB2.0    CardReader SM XD  0100  /dev/sdc
[2:0:0:2]    disk     USB2.0    CardReader MS     0100  /dev/sdd
[2:0:0:3]    disk     USB2.0    CardReader SD     0100  /dev/sde
[3:0:0:0]    disk     FLASH     Drive UT_USB20    0.00  /dev/sdf
```

方括号中的数字是SCSI主机适配器编号，SCSI总线编号，设备SCSI ID，以及LUN（逻辑元件编号，设备的字设备）。本例中有4个适配器（scsi0、scsi1、scsi2和scsi3），它们都有一个单独的总线（总线编号都是0），每个总线上有一个设备（target编号都是0）。编号为2:0:0的USB读卡器有4个逻辑单元，每个代表一个可插入闪存盘。内核为每个逻辑单元指定一个不同的设备文件。

图3-2显示内核中该部分的驱动和接口程序结构，从单个设备驱动上到块设备驱动，但是不包括SCSI通用驱动。

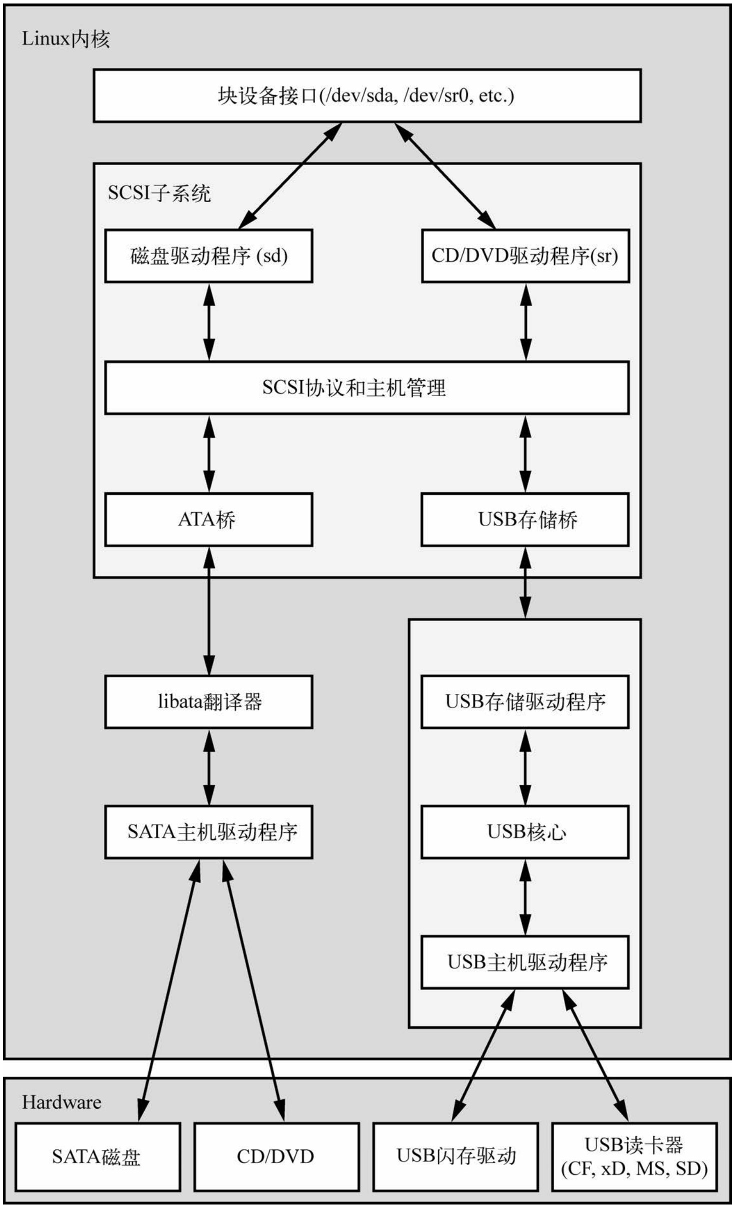


图3-2　Linux SCSI 子系统示意图

上图看上去复杂，实际上整个结构是非常线性的。我们先从SCSI子系统和它的三层驱动开始。

- 最顶层负责处理某一类设备。例如，sd（SCSI硬盘）驱动就在这一层，它负责将来自内核块设备接口的请求消息翻译为SCSI协议中的硬盘相关命令，反之亦然。
- 中间层在上下层之间调控和分流SCSI消息，并且负责管理系统中的所有SCSI总线和设备。

- 最底层负责处理硬件相关操作。该层中的驱动程序向特定的主机适配器发送SCSI协议消息，并且提取从硬件发送过来的消息。该层和最顶层分开的原因是，虽然SCSI消息对某类设备是统一的，但是不同类型的主机适配器处理同类消息的方式会不一样。

最顶层和最底层中有许多各式各样的驱动程序，但是需要注意，对每一个设备文件，内核都使用一个顶层中的驱动程序和一个底层中的驱动程序。对我们例子中的/dev/sda硬盘来说，内核使用顶层的sd和底层的ATA bridge。

有时候你可能需要使用一个以上的顶层驱动程序（见3.6.3节）。对于真正的SCSI设备，如连接到SCSI主机适配器或者硬件RAID的硬盘，底层驱动程序直接和下方的硬件通信，这与大部分SCSI子系统中的设备不同。

3.6.1　USB存储设备和SCSI

如图3-2所示，内核需要更多那样的底层SCSI驱动来支持SCSI子系统和USB存储设备硬件的通信。/dev/sdf代表的USB闪存驱动支持SCSI命令，但是不和驱动通信，所以由内核来负责和USB系统的通信。

理论上，USB和SCSI很类似，包括设备类别、总线、主机控制器。所以和SCSI类似，Linux内核也有一个三层USB子系统。最顶层是同类设备驱动，中间层是总线管理，最底层是主机控制驱动。和SCSI类似，USB子系统通过USB消息在其组件之间通信，它还有一个和`lsscsi`类似的命名叫`lsusb`。

最顶层是我们介绍的重点，在这里驱动程序如同一个翻译，它对一方使用SCSI协议通信，对另一方使用USB协议，并且存储硬件在USB消息中包含了SCSI命令，所以启动程序要做的翻译工作仅仅是重新打包消息数据。

有了SCSI和USB子系统，你就能够和闪存驱动通信了。还有不要忘了SCSI子系统更底层的驱动程序，因为USB存储驱动是USB子系统的一部分，而非SCSI子系统。（出于某些原因，两个子系统不能共享驱动程序）。如果一个子系统要和其他子系统通信，需要使用一个简单的底层SCSI桥接驱动来连接USB子系统的存储驱动程序。

3.6.2　SCSI和ATA

图3-2中的SATA硬盘和光驱使用的都是SATA接口。和USB驱动一样，内核需要一个桥接驱动来将SATA驱动连接到SCSI子系统，不过使用的是另外的更复杂的方式。光驱使用ATAPI协议通信，它是使用了ATA协议编码的一种SCSI命令。然而硬盘不使用ATAPI和编码的SCSI命令。

Linux内核使用libata库来协调SATA（以及ATA）驱动和SCSI子系统。对于支持ATAPI的光驱，问题变得很简单，只需要提取和打包往来于ATA协议上的SCSI命令即可。对于硬盘就复杂得多，libata库需要一整套命令翻译机制。

光驱的作用类似于把一本英文书敲入计算机。你不需要了解书的内容，甚至不懂英文也没关系。硬盘的工作则类似把一本德文书翻译成英文并敲入计算机。所以你必须懂两种语言，以及了

解书的内容。

libata能够将SCSI子系统连接到ATA/SATA接口和设备。（为了简单起见，图3-2只包含了一个SATA主机驱动，实际上不止一个驱动。）

3.6.3　通用SCSI设备

用户空间进程和SCSI子系统的通信通常是通过块设备层和（或者）在SCSI设备类驱动之上的另一个内核服务（如sd或者sr）来进行。换句话说，大多数用户进程不需要了解SCSI设备和命令。

然而，用户进程也可以绕过设备类驱动通过*通用设备*和SCSI设备直接通信。例如我们在3.6节介绍过的，我们使用`lsscsi`的-g选项来显示通用设备，结果如下：

```
$ lsscsi -g
[0:0:0:0]    disk    ATA      WDC WD3200AAJS-2 01.0  /dev/sda ❶/dev/sg0
[1:0:0:0]    cd/dvd  Slimtype DVD A DS8A5SH    XA15  /dev/sr0   /dev/sg1
[2:0:0:0]    disk    USB2.0   CardReader CF    0100  /dev/sdb   /dev/sg2
[2:0:0:1]    disk    USB2.0   CardReader SM XD 0100  /dev/sdc   /dev/sg3
[2:0:0:2]    disk    USB2.0   CardReader MS    0100  /dev/sdd   /dev/sg4
[2:0:0:3]    disk    USB2.0   CardReader SD    0100  /dev/sde   /dev/sg5
[3:0:0:0]    disk    FLASH    Drive UT_USB20   0.00  /dev/sdf   /dev/sg6
```

除了常见的块设备文件，上面的每一行还在❶字段位置显示SCSI通用设备文件。例如光驱/dev/sr0的通用设备是/dev/sg1。

那么我们为什么需要SCSI通用设备呢？原因来自内核代码的复杂度。当任务变得越来越复杂的时候，最好是将其从内核移出来。我们可以考虑下CD/DVD的读写操作，写数据操作比读复杂得多，并且没有任何关键的系统服务需要依赖于CD/DVD的写数据操作。使用用户空间进程来写数据也许比使用内核服务要慢，但是却更容易开发和维护，并且如果有bug也不会影响到内核空间。所以在向CD/DVD写数据时，进程就使用像/dev/sg1这样的通用SCSI设备。至于读取数据，虽然很简单，但我们仍然使用内核中一个特制的sr光驱驱动来完成。

3.6.4　访问设备的多种方法

图3-3展示了从用户空间访问光驱的两种方法：sr和sg（图中忽略了在SCSI更下层的驱动）。进程A使用sr驱动来读数据，进程B使用sg驱动。然而，它们之间并不是并行访问的。

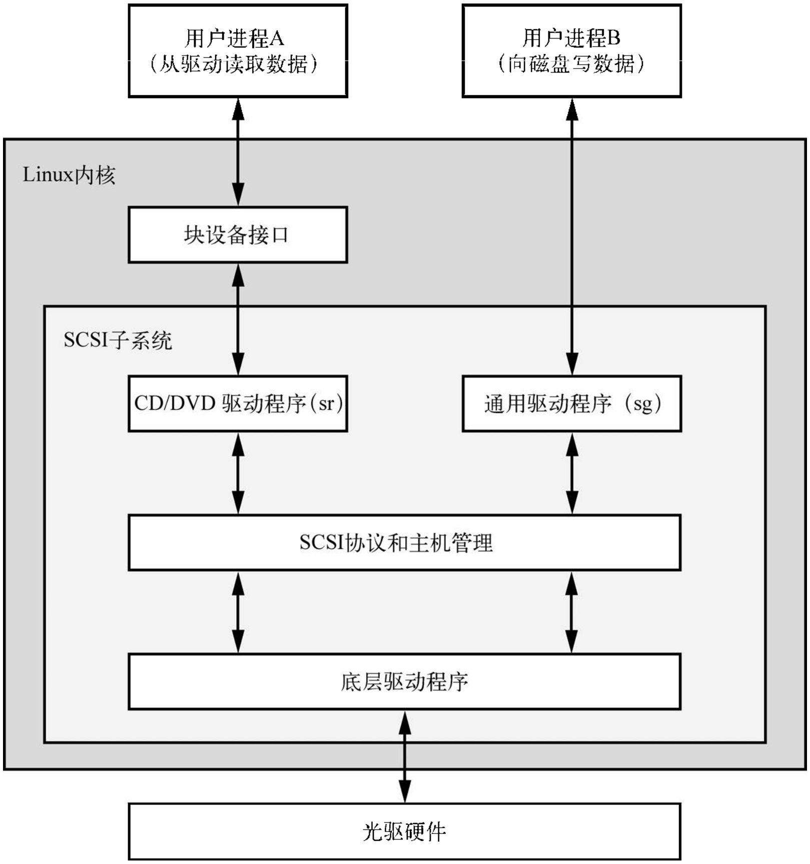
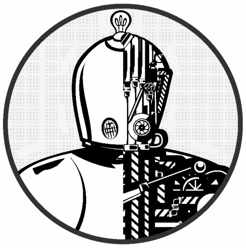


图3-3　光学设备图解

图中进程A从块设备读取数据，但是通常用户进程不使用这样的方式读取数据，至少不是直接读取。在块设备之上还有很多的层和访问入口，我们将在下一章介绍。

第 4 章

硬盘和文件系统



在第3章中我们讨论了内核提供的顶层磁盘设备。本章我们将详细介绍如何在Linux系统中使用磁盘设备。你将了解如何为磁盘分区，在分区中创建和维护文件系统，以及交换空间。

磁盘设备对应/dev/sda这样的设备文件，它代表SCSI子系统中的第一个磁盘。诸如此类的块设备代表整块磁盘，磁盘中又包含很多不同的组件和层。

图4-1显示了典型的Linux磁盘的大致结构，本章将逐一介绍其中的各个部分。

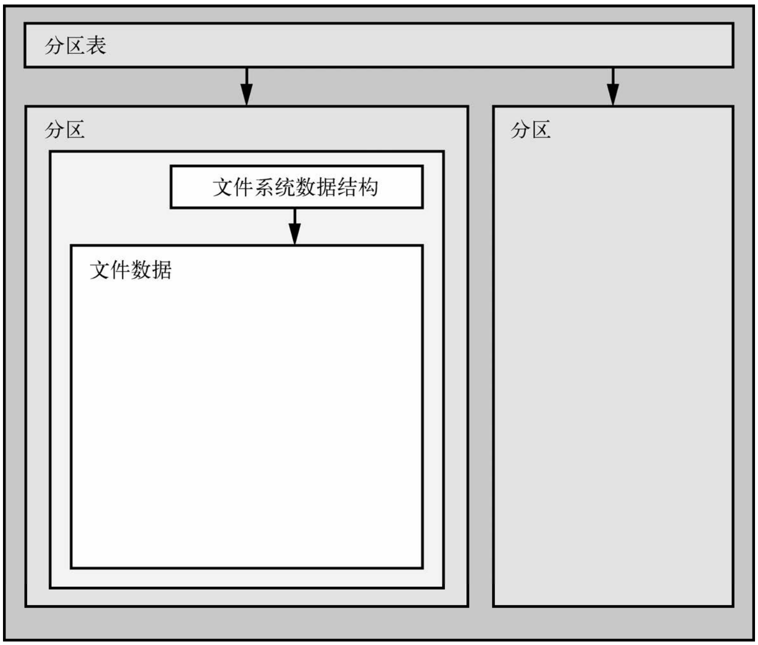


图4-1　Linux磁盘图解

分区是对整块磁盘的进一步划分，在Linux系统中由磁盘名称加数字来表示，比如/dev/sda1和/dev/sdb3。内核将分区用块设备呈现，如同每个分区是一整块的磁盘。分区数据存放在磁盘上的分区表中。

注解　通常我们会在一块大磁盘上划分多个分区，因为老的计算机系统只能使用磁盘上的特定部分来启动。系统管理员也通过分区来为操作系统预留一定的空间。例如，管理员不希望用户用满整个磁盘从而导致系统无法工作。这些做法不只见于Unix，Windows也是这样。此外，大部分的系统还有一个单独的交换分区。

虽然内核允许你同时访问整块磁盘和某个分区，但是一般不需要这样做，除非你在复制整个磁盘。

分区之下的一层是文件系统，文件系统是用户空间中与你日常交互的文件和目录数据库。我们将在4.2节中详细介绍。

如图4-1所示，如果你想要访问文件中的数据，你需要从分区表中获得分区所在位置，然后在该分区的文件系统数据库中查找指定文件的数据。

Linux内核使用图4-2中的各个层来访问磁盘上的数据。SCSI子系统和我们在3.6节中介绍的内容由一个框代表。（请注意，你可以通过文件系统或者磁盘设备来访问磁盘，我们本章都将介绍。）

让我们从最底部的分区开始。

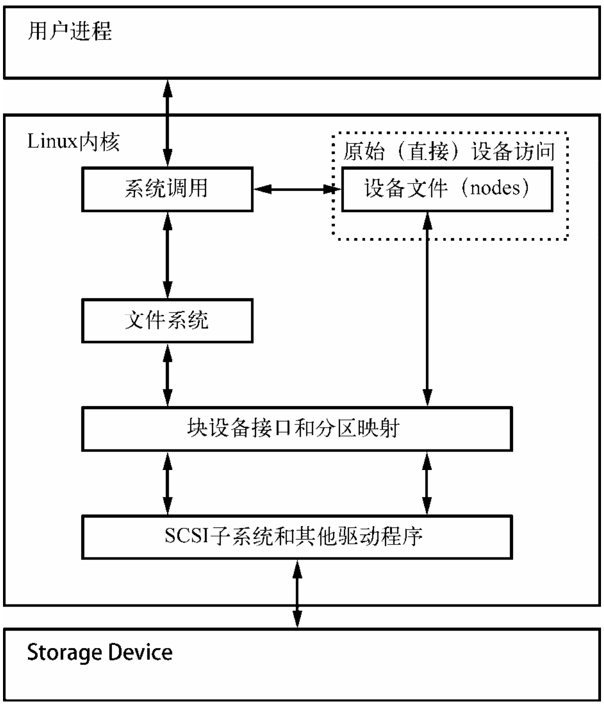


图4-2　内核磁盘访问图解

4.1　为磁盘设备分区

分区表有很多种，比较典型的一种叫主引导记录（Master Boot Record，以下简称MBR）。另一种逐渐普及的是全局唯一标识符分区表（Globally Unique Identifier Partition Table，以下简称GPT）。

下面是Linux系统中的各种分区工具。

- parted：一个文本命令工具，支持MBR和GPT。
- gparted：parted的图形版本。
- fdisk：Linux传统的文本命令分区工具，不支持GPT。
- gdisk：fdisk的另一个版本，支持GPT，但不支持MBR。

虽然很多人喜欢使用fdisk，但本书将着重介绍支持MBR和GPT的parted。

> **注解**　parted虽然也能够创建和操作文件系统，但是你最好不要使用它来操作文件系统，因为这样会引发一些混淆。分区操作和文件系统操作还是有本质的不同。分区表划分磁盘的区域，而文件系统侧重数据管理，因此我们使用parted分区，使用另外的工具来创建文件系统（见4.2.2节），parted的文档中也是这样建议的。

4.1.1　查看分区表

你可以使用命令parted -l查看系统分区表。如下例所示。

```
# parted -l
Model: ATA WDC WD3200AAJS-2 (scsi)
Disk /dev/sda: 320GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number  Start   End     Size    Type      File system     Flags
 1      1049kB  316GB   316GB   primary   ext4            boot
 2      316GB   320GB   4235MB  extended
 5      316GB   320GB   4235MB  logical   linux-swap(v1)

Model: FLASH Drive UT_USB20 (scsi)
Disk /dev/sdf: 4041MB
Sector size (logical/physical): 512B/512B
Partition Table: gpt

Number  Start   End     Size    File system  Name      Flags
 1      17.4kB  1000MB  1000MB               myfirst
 2      1000MB  4040MB  3040MB               mysecond
```

第一个设备/dev/sda使用传统的MBR分区表（parted中称为msdos），第二个设备使用GPT表。

请注意，由于分区表类型不同，所以它们的参数也不同。MBR表中没有名称这一列，而GPT表中则有名称列（这里我们随意取了两个名称myfirst和mysecond）。

上例中的MBR表包含主分区、扩展分区和逻辑分区。主分区是磁盘的常规分区（如上例中的1）。MBR最多只能有4个主分区，如果需要更多分区，你要将一个分区设置为扩展分区，然后将该扩展分区划分为数个逻辑分区。上例中的2是扩展分区，在其上有逻辑分区5。

注解 parted命令显示的文件系统不一定是MBR中的系统ID。MBR系统ID只是一个数字，例如83是Linux分区，82是Linux交换分区。因而parted自己来决定文件系统。如果你想知道MBR中的系统ID，可以使用命令fdisk -l。

内核初始化读取

Linux内核在初始化读取MBR表时，会显示以下的调试信息（使用dmesg命令来查看）：

```
sda: sda1 sda2 < sda5 >
```

sda2 < sda5 >表示/dev/sda2是一个扩展分区，它包含一个逻辑分区/dev/sda5。通常你只需要访问逻辑分区，所以扩展分区可以忽略。

4.1.2　更改分区表

查看分区表相对更改分区表来说比较简单和安全，虽然更改分区表也不是很复杂，但还是有一定的风险，所以需要特别注意以下两点。

- 删除分区以后，分区上的数据很难被恢复，因为你删除的是该文件系统最基本的信息。所以最好事先对数据做备份。
- 确保你操作的磁盘上没有分区正在被系统使用。因为大多数Linux系统会自动挂载被删除的文件系统（参见4.2.3节）。

一切准备就绪后，你可以开始选择使用哪个分区程序了。你可以使用命令行工具parted或者图形界面工具gparted。如果你在使用GPT分区的话，可以使用gdisk。这些工具都有在线帮助，很容易掌握。（如果你的磁盘空间不够，你可以使用闪存盘等设备来尝试使用它们。）

fidsk和parted有很大区别。fdisk让你首先设计好分区表，然后在退出fdisk之前才做实际的更改。parted则是在你运行命令的同时直接执行创建、更改和删除操作，你没有机会在做更改之前确认检查。

这样的区别也能够帮助我们了解它们和内核是如何交互的。fdisk和parted都是在用户空间中对分区做更改，所以没有必要为它们提供内核支持。

但是，内核还是必须负责读取分区表并将分区呈现为块设备。fdisk使用了一种相对简单的方式来处理：更改分区表之后，fdisk向内核发送一个磁盘系统调用，告诉内核需要重新读取分区表，内核会显示一些调试信息供你使用dmesg查看。例如，如果你在/dev/sdf上创建了两个分区，

你会看到如下信息：

```
sdf: sdf1 sdf2
```

相比之下，parted没有使用磁盘系统调用，而是在分区表被更改的时候向内核发送信号，内核也不显示调试信息。

你可以用以下方式来查看对分区的更改。

- 使用udevadm查看内核消息更改。例如：udevadm monitor --kernel会显示被删除的分区和新创建的分区。
- 在/proc/partitions中查看完整的分区信息。
- 在/sys/block/device中查看更改后的分区系统接口信息，在/dev中查看更改后的分区设备。

如果你想100%确定你是否更改了分区表，你可以使用blockdev命令。例如，要让内核重新加载/dev/sdf上的分区表，可以运行下面的命令：

```
# blockdev --rereadpt /dev/sdf
```

到目前为止，你应该了解了磁盘分区的相关内容，如果你想了解更多有关磁盘的内容，可以继续。你也可以直接跳到4.2节去了解文件系统的内容。

4.1.3　磁盘和分区的构造

包含可移动部件的设备会为操作系统带来一定的复杂度，因为抽象起来比较复杂。磁盘就是这样的设备，虽然你可以把它看成是一个块设备，可以随机访问其中的任何地方，但是如果你对磁盘数据规划得不好，就会导致很严重的性能问题。参见图4-3。

磁盘中有一个转轴，上面有一个转盘，还有一个覆盖磁盘半径的可以移动的杆，上面有一个读写头，磁盘在读写头下旋转，读写头负责读取数据。读写头只能读取移动杆当前所在位置圆周范围内的数据。这个圆周我们称作*柱面*。大容量的磁盘通常在一个转轴上有多个转盘叠加在一起旋转。每个转盘有一到两个读写头，负责转盘正面和背面的读写。所有读写头都由一个移动杆控制，协同工作。磁盘的柱面从圆心到边缘由小变大。你可以将柱面划分为*扇区*。磁盘这样的构造我们称为CHS，意指柱面（cylinder）–读写头（head）–扇区（sector）。

> **注解**　*磁盘轨道是读写头访问柱面的那一部分，在图4-3中，柱面也是一个轨道。对磁盘轨道你可以不用深究。*

通过内核和各种分区程序你能够知道磁盘的柱面（和扇区）数。然而在现在的硬盘中这些数字*并不准确*。传统使用CHS的寻址方式无法适应现在的大容量硬盘，也无法处理外道柱面比内道柱面存储更多数据这样的情况。磁盘硬件支持逻辑块寻址（Logical Block Addressing，以下简称LBA），通过块编号来寻址，其余部分还是使用CHS。例如，MBR分区表包含CHS信息和对应的

LBA信息，虽然一些引导装载程序仍然使用CHS，但大部分都是用LBA。

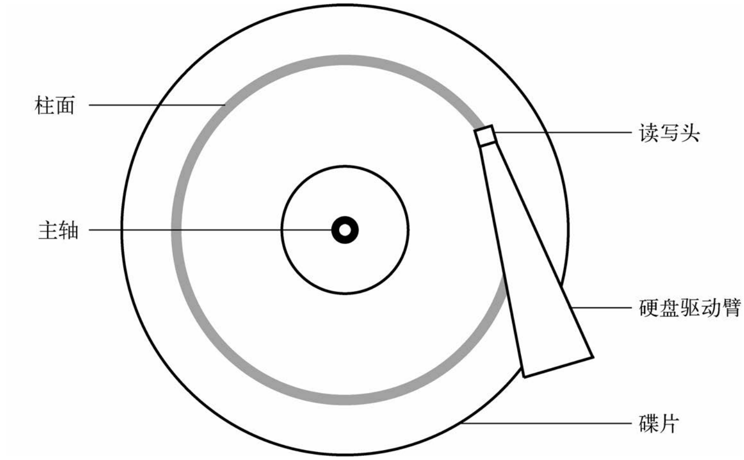


图4-3　硬盘俯视图

柱面是一个很重要的概念，用来设置分区边界。从柱面读取数据速度是很快的，因为磁盘的旋转可以让读写头持续地读取数据。将分区放到相邻柱面也能够加快数据存取，因为这样可以缩小读写头移动的距离。

如果你没有将分区精确地置于柱面边界，一些分区程序会提出警告，你可以忽略之，因为现在的硬盘提供的CHS信息不准确。LBA则能够确保你的分区位置正确。

4.1.4　固态硬盘

固态硬盘（Solid State Disk，以下简称SSD）这样的存储设备和旋转式硬盘区别很大，因为它们没有可移动的部件。由于不需要读写头在柱面上移动，所以随机存取不成问题，但是也有一些因素影响到其性能。

分区布局是影响SSD性能的一个方面。SSD通常每次读取4096字节的数据，所以如果要读取的数据没有在同一区块内，就需要两次读取操作。这对于那些经常性的操作（如读取目录内容）来说性能会降低。

许多分区工具（如parted和gparted）能够为新分区设置合理的位置，所以你不需要担心这个问题。不过如果你想知道你的分区所在位置和边界，例如分区/dev/sdf2，你可以使用以下命令查看：

```
$ cat /sys/block/sdf/sdf2/start
1953126
```

该分区从距离磁盘起始位置1 953 126字节的地方开始，由于不是4096的整数倍，所以该分区在SSD上无法获得最佳性能。

4.2 文件系统

文件系统通常是内核和用户空间之间联系的最后一环，也就是通过ls和cd等命令进行交互的对象。之前介绍过，文件系统是一个数据库，它将简单的块设备映射为用户易于理解的树状文件目录结构。

以往的文件系统位于磁盘和其他类似的存储设备上，只单纯地负责数据存储。然而文件系统的树状文件目录结构和I/O接口还有很多功能。现在的文件系统能够处理各式各样的任务，比如目录/sys和/proc中的那些系统接口。以往都是由内核负责实现文件系统，Plan 9的9P（http://plan9.bell-labs.com/sys/doc/9.html）的出现促进了文件系统在用户空间中的实现。用户空间文件系统（File System in User Space，简称FUSE）这一特性使得在Linux上实现用户空间文件系统成为可能。

4

抽象层虚拟文件系统（Virtual File System，以下简称VFS）负责文件系统的具体实现。像SCSI子系统将设备之间和设备与内核之间的通信标准化一样，VFS为用户空间进程访问不同文件系统提供了标准的接口。VFS使得Linux能够支持很多不同的文件系统。

4.2.1 文件系统类型

Linux支持原生设计的并且针对Linux进行过优化的文件系统，支持Windows FAT这样的外来文件系统，支持ISO 9660这样的通用文件系统，以及很多其他文件系统。下面我们列出了常见的文件系统，Linux能够识别的那些我们在名称后加上了类型名称和括号。

- 第四扩展文件系统（以下简称ext4）：是Linux原生文件系统的当前版本。第二扩展文件系统（以下简称ext2）作为Linux的默认系统已经存在了很长时间，它源于传统的Unix文件系统（如Unix File System - UFS和Fast File System - FFS）。第三扩展文件系统（以下简称ext3）增加了日志特性（在文件系统数据结构之外的一个小的缓存机制），提高了数据的完整性和启动速度。ext4文件系统在ext2和ext3的基础上不断完善，支持更大的文件和更多的子目录个数。扩展文件系统的各个版本都向后兼容。例如，你可以将ext2和ext3挂载为ext3和ext2，你也可以将ext2和ext3挂载为ext4，但是你不能将ext4挂载为ext2和ext3。
- ISO 9660（iso9660）：是一个CD-ROM标准。大多数CD-ROM都是使用该标准的某个版本。
- FAT文件系统（msdos、vfat、umsdos）：是微软的文件系统。msdos很简单，支持最基本的单字符MS-DOS系统。在新版本的Windows中如果要支持Linux，你应该使用vfat文件系统。umsdos这个系统很少用到，它在MS-DOS的基础上支持Linux和一些Unix特性，如符号链接。
- HFS+（hfsplus）：是苹果Macintosh计算机的文件系统标准。

虽然扩展文件系统的各个版本已经应用得很好，然而其中也加入了很多高级的功能，ext4由于向后兼容的考虑都无法使用。高级功能主要涉及对大数量文件、大尺寸文件以及其他类似情形的支持。正在开发中的Btrfs这样的文件系统将来有可能取代扩展文件系统。

4.2.2　创建文件系统

当完成了4.1节中介绍的分区操作后，你就可以创建文件系统了。和分区一样，用户空间进程能够访问和操作块设备，所以你可以在用户空间中创建文件系统。mkfs工具可以创建很多种文件系统，例如，你可以使用以下命令在/dev/sdf2上创建ext4分区：

```
# mkfs -t ext4 /dev/sdf2
```

mkfs能够自己决定设备上的块数量并且设置适当的默认值，除非你确定该怎么做并且阅读了详细的文档，否则你不需要更改默认参数。

mkfs在创建文件系统的过程中会显示诊断信息，其中包括*超级块*的输出信息。超级块是文件系统数据库上层的一个重要组件，以至于mkfs对其有多个备份以防其损坏。你可以记录下超级块备份编号以防万一磁盘出现故障（参见4.2.11节）。

警告　你只需要在增加新磁盘和修复现有磁盘的时候创建文件系统。一般是对没有数据的新分区进行此操作，或者分区已有的数据你不再需要了。如果在已有文件系统上创建新的文件系统，所有已有的数据将会丢失。

mkfs是一系列文件系统创建程序的前端界面，如mkfs.fs。fs是一种文件系统类型。当运行mkfs -t ext4时，实际上运行的是mkfs.ext4。

你可以通过查看mkfs.*文件看到更多的相关程序：

```
$ ls -l /sbin/mkfs.*
-rwxr-xr-x 1 root root 17896 Mar 29 21:49 /sbin/mkfs.bfs
-rwxr-xr-x 1 root root 30280 Mar 29 21:49 /sbin/mkfs.cramfs
lrwxrwxrwx 1 root root     6 Mar 30 13:25 /sbin/mkfs.ext2 -> mke2fs
lrwxrwxrwx 1 root root     6 Mar 30 13:25 /sbin/mkfs.ext3 -> mke2fs
lrwxrwxrwx 1 root root     6 Mar 30 13:25 /sbin/mkfs.ext4 -> mke2fs
lrwxrwxrwx 1 root root     6 Mar 30 13:25 /sbin/mkfs.ext4dev -> mke2fs
-rwxr-xr-x 1 root root 26200 Mar 29 21:49 /sbin/mkfs.minix
lrwxrwxrwx 1 root root     7 Dec 19  2011 /sbin/mkfs.msdos -> mkdosfs
lrwxrwxrwx 1 root root     6 Mar 5   2012 /sbin/mkfs.ntfs -> mkntfs
lrwxrwxrwx 1 root root     7 Dec 19  2011 /sbin/mkfs.vfat -> mkdosfs
```

如你所见，mkfs.ext4只是mke2fs的一个符号链接。如果你在系统中没有发现某个特定的mkfs命令，或者你在寻找某个特定的文件系统类型，记住这点很重要。文件系统创建程序都有自己的使用手册，如mke2fs(8)，在大部分情况下运行mkfs.ext4(8)将会重定向到mke2fs(8)。

4.2.3　挂载文件系统

在Unix系统中，我们称挂载文件系统为mounting。系统启动的时候，内核根据配置信息挂载root目录/。

要挂载文件系统，你需要了解以下几点。

- 文件系统所在设备（如磁盘分区，文件系统数据存放的位置）。
- 文件系统类型。
- 挂载点，也就是当前系统目录结构中挂载文件系统的那个位置。挂载点是一个目录，例如，你可以使用/cdrom目录来挂载CD-ROM。挂载点可以在任何位置，只要不直接在/下即可。

挂载文件系统时我们常这样描述："将x设备挂载到x挂载点。"你可以运行mount命令来查看当前文件系统状态，如：


```
$ mount
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
none on /sys/fs/fuse/connections type fusectl (rw)
none on /sys/kernel/debug type debugfs (rw)
none on /sys/kernel/security type securityfs (rw)
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
--snip--
```

上面每一行对应一个已挂载的文件系统，每一列代表的信息如下所示。

- 设备名，如/dev/sda3。其中有一些不是真实的设备（如proc），它们代表一些特殊用途的文件系统，不需要实际的设备。
- 单词on。
- 挂载点。
- 单词type。
- 文件系统类型，通常是一个短标识。
- 挂载选项（使用括号包围，详见4.2.6节）。

使用mount命令带参数（文件系统类别、设备及所需的挂载点）来挂载文件系统，如下所示：

```
# mount -t type device mountpoint
```

例如要挂载/dev/sdf2设备到/home/extra，可以运行下面的命令：

```
# mount -t ext4 /dev/sdf2 /home/extra
```

一般情况下不需要指定-t type参数，mount命令可以自行判断。然而有时候需要在类似的文件系统中明确指定一个，比如不同的FAT文件系统类型。

在4.2.6节我们将介绍更多选项。至于文件系统的卸载，你可以使用umount命令：

```
# umount mountpoint
```

你也可以卸载设备而不是挂载点。

4.2.4　文件系统UUID

上一节介绍的挂载文件系统使用的是设备名。然而设备名会根据内核发现设备的顺序而改变，因此你可以使用文件系统的通用唯一标识（Universally Unique Identifier，以下简称UUID）来挂载。UUID是一系列的数字，并且保证每个唯一。mke2fs这些文件系统创建程序在初始化文件系统数据结构时会生成一个UUID。

你可以使用blkid（block ID）命令查看设备和其对应的文件系统及UUID：

```
# blkid
/dev/sdf2: UUID="a9011c2b-1c03-4288-b3fe-8ba961ab0898" TYPE="ext4"
/dev/sda1: UUID="70ccd6e7-6ae6-44f6-812c-51aab8036d29" TYPE="ext4"
/dev/sda5: UUID="592dcfd1-58da-4769-9ea8-5f412a896980" TYPE="swap"
/dev/sde1: SEC_TYPE="msdos" UUID="3762-6138" TYPE="vfat"
```

上例中blkid发现了四个分区，其中有2个ext4、1个交换分区（见4.3节）以及1个FAT。Linux原生分区都有标准UUID，但是FAT分区没有。FAT分区可以通过FAT 卷序列号（本例中是3762-6138）来引用。

使用UUID=来通过UUID挂载文件系统。例如，要把上例中的第一个文件系统挂在到/home/extra，可以运行如下命令：

```
# mount UUID=a9011c2b-1c03-4288-b3fe-8ba961ab0898 /home/extra
```

通常你会使用设备名来挂载文件系统，因为这比UUID容易。但是理解UUID也非常重要，因为系统启动时（见4.2.8节）倾向于使用UUID来挂载文件系统。此外，很多Linux系统使用UUID作为可移动媒体的挂载点。上例中的FAT文件系统就是在一个闪存卡上。Ubuntu系统会在该设备插入时将这个分区挂载为/media/3762-6138。udevd守护进程（见第3章）负责处理设备插入的初始事件。

必要时你可以更改文件系统的UUID（例如你拷贝一个文件系统后，需要区分原来的和新拷贝的文件时）。有关更改ext2/ext3/ext4的UUID的内容，你可以参阅tune2fs(8)使用手册。

4.2.5　磁盘缓冲、缓存和文件系统

Linux和其他Unix系统一样，将写到磁盘的数据先写入缓冲区。这意味着内核在处理更改请求的时候不直接将更改写到文件系统，而是将更改保存到RAM中直到内核能够便捷地将更改写到磁盘为止。这个缓冲机制能够带来性能上的提高，对用户来说是透明的。

当你使用umount来卸载文件系统时，内核自动和磁盘同步。另外你还可以随时使用sync命令强制内核将缓冲区的数据写到磁盘。如果你在关闭系统之前由于种种原因无法卸载文件系统，请务必先运行sync命令。

此外，内核有一系列的机制使用RAM自动缓存从磁盘读取的数据块。因而对重复访问同一个

文件的多个进程来说，内核不用反复地读取磁盘，而只需要从缓存中读取数据来节省时间和资源。

4.2.6 文件系统挂载选项

mount命令的更改有很多选项，数量还不少，通常在处理可移动设备的时候需要用到。mount(8)使用手册提供了详细的参考信息，但是你很难从中找出哪些应该掌握，哪些可以忽略。本节我们介绍一些比较有用的选项。

这些选项大致可以分为两类：通用类和文件系统相关类。通用选项有-t（指定文件系统类型，前文介绍过）。文件系统相关选项只对特定的文件系统类型适用。

文件系统相关选项使用方法是在-o后加选项。例如，-o norock在ISO 9660文件系统上关闭Rock Ridge扩展，但是该选项对其他文件系统类型无效。

短选项

以下是一些比较重要的通用选项。

- -r：该选项以只读模式挂载文件系统，应用在许多场景，如写保护和系统启动。在挂载只读设备（如CD-ROM）的时候，可以不需要设置该选项，系统会自动设置（还会提供只读设备状态）。
- -n：该选项确保mount命令不会更新系统运行时的挂载数据库/etc/mtab。如果无法成功写这个文件，mount命令就会失败。因为系统启动时root分区（存放系统挂载数据库的地方）最开始是只读的，所以这个选项十分重要。在单用户模式下修复系统问题时这个选项也很有用，因为系统挂载数据库也许在那时会不可用。
- -t：-t type选项指定文件系统类型。

长选项

短选项对越来越多的挂载选项来说明显不够用了，一是26个字母无法容纳所有选项，二是单个字母很难说明选项的功能。很多通用选项和文件系统相关选项都更长，格式也更灵活。

长选项的使用方法是在-o后加关键字，见下例：

```
# mount -t vfat /dev/hda1 /dos -o ro,conv=auto
```

ro和conv=auto是两个长选项。ro和-r一样，设定只读模式。conv=auto告诉内核自动将文本文件从DOS格式转换为Unix格式（稍后详细介绍）。

以下是比较常用的长选项。

- exec、noexec：允许和禁止在文件系统上执行程序。
- suid、nosuid：允许和禁止setuid程序。
- ro：在只读模式下挂载文件系统（同-r）。
- rw：在读写模式下挂载文件系统。
- conv=rule（FAT文件系统）：根据rule规则转换文件中的换行符，rule的值为binary、text或auto，默认为binary。binary选项禁止任何字符转换。text选项将所有文件当作文本文

件。auto选项根据文件扩展名来进行转换。例如，对.jpg文件不做任何处理，而对.txt文件则进行转换。使用这个选项时需要谨慎，因为它可能对文件造成损坏，可以考虑在只读模式中使用。

4.2.7　重新挂载文件系统

有时候你可能由于需要更改挂载选项而在同一挂载点重新挂载文件系统。比较常见的情况是在崩溃恢复时你需要将只读文件系统改为可写。

以下命令以可读写模式重新挂载root（你需要-n选项，因为mount命令在root为只读的情况下无法写系统挂载数据库）：

```
# mount -n -o remount /
```

该命令假定/设备在目录/etc/fstab中（下节将会介绍），否则你需要指定设备。

4.2.8　/etc/fstab文件系统表

为了在系统启动时挂载文件系统和降低mount命令的使用，Linux系统在/etc/fstab中永久保存了文件系统和选项列表。它是一个纯文本文件，格式很简单，如例4-1所示。

例4-1　/etc/fstab中的文件系统和选项列表

```
proc /proc proc nodev,noexec,nosuid 0 0
UUID=70ccd6e7-6ae6-44f6-812c-51aab8036d29 / ext4 errors=remount-ro 0 1
UUID=592dcfd1-58da-4769-9ea8-5f412a896980 none swap sw 0 0
/dev/sr0 /cdrom iso9660  ro,user,nosuid,noauto 0 0
```

其中每一行对应一个文件系统，且每一行有6个字段，从左至右分别如下所示。

- 设备或者UUID：最新版本的Linux系统不再使用/etc/fstab中的设备，而是使用UUID。（请注意，/proc这一行有一个名为proc的象征性设备。）
- 挂载点：指定挂载文件系统的位置。
- 文件系统类型：你可能在列表中没发现swap字样，它代表交换分区（见4.3节）。
- 选项：使用逗号作为长选项的分隔符。
- 提供给dump命令使用的备份信息：你应该总是使用0。
- 文件系统完整性测试顺序：为了确保fsck总是第一个在root上运行，对root文件系统总是将其设置为1，硬盘上的其他文件系统设置为2。使用0来禁止其他启动检查，包括CD-ROM、交换分区和/proc文件系统（见4.2.11节）。

使用mount时，如果你操作的文件系统在/etc/fstab中的话，你可以使用一些快捷方式。例如，如果你在使用例4-1挂载CD-ROM，你可以运行mount /cdrom。

使用以下命令，你可以挂载/etc/fstab中的所有未标识为noauto的设备：

```
# mount -a
```

例4-1中有一些新选项，如errors、noauto和user，因为它们只在/etc/fstab文件中适用。另外，你会经常看到defaults选项。它们各自代表的意思如下所示。

- defaults：该选项使用mount的默认值——读写模式、启动设备文件、可执行、setuid等等。如果你不想对任何列设置特殊值的话就使用该选项。
- errors：这是ext2相关参数，用来定义内核在系统挂载出现问题时的行为。默认值通常是errors=continue，意指内核应该返回错误代码并且继续运行。errors=remount-ro是告诉内核以只读模式重新挂载。errors=panic使内核在挂载发生问题时停止。
- noauto：该选项让mount -a命令忽略本行设备。可以使用这个选项来防止在系统启动时挂载可移动设备如CD-ROM和软盘。
- user：这个选项能够让没有权限的用户对某一行设备运行mount命令，对访问CD-ROM这些设备来说比较方便。因为用户可以通过其他系统在移动设备上放置一个setuid-root文件，这个选项也设置nosuid、noexec和nodev（用来阻止特殊的设备文件）。

4.2.9　/etc/fstab的替代者

一直以来我们都使用/etc/fstab来管理文件系统和挂载点，但也有两种新的方式。一是/etc/fstab.d目录，其中包含了各个文件系统的配置文件（每个文件系统有一个文件）。该目录和本书中你见到的其他配置文件目录非常类似。

另一种方式是为文件系统配置systemd单元。我们将在第6章介绍systemd及其单元。不过，systemd单元配置通常是由或者说基于/etc/fstab生成的，所以你可能会发现一些重叠的部分。

4.2.10　文件系统容量

你可以使用df命令查看当前挂载的文件系统的容量和使用量，如下例所示：

```
$ df
Filesystem    1024-blocks      Used   Available   Capacity Mounted on
/dev/sda1         1011928     71400      889124         7%  /
/dev/sda3        17710044   9485296     7325108        56%  /usr
```

我们来看看df命令输出的各字段的含义。

- Filesystem：指文件系统设备。
- 1024-blocks：指文件系统的总容量，以每块1024字节为单位。
- Used：指已经使用的容量。
- Available：指剩余的容量。
- Capacity：指已经使用容量的百分比。
- Mounted on：指挂载点。

显而易见，上例中两个文件系统容量分别为1 GB和17.5 GB。然而容量数据可能看起来有些奇怪，因为71 400加889 124不等于1 011 928，9 485 296也不是17 710 044的56%。这是因为两个

文件系统中各有总容量的5%没被计算在内，它们是隐藏的预留块。所以只有超级用户在需要时能够使用全部的空间。这个特性使得磁盘空间被占满后系统不会马上崩溃。

如果你的磁盘空间满了，你想查看哪些文件占用了大量空间，可以使用du命令。如果不带任何参数，du将显示当前工作目录下的所有目录的磁盘使用量。（你可以运行cd /; du试试看，如果你看够了可以按Ctrl+C。）du -s命令只显示合计数。如果想查看某个特定目录的容量，可以切换到该目录，运行du -s *来查看结果。

注解　POSIX标准中定义每个块大小是512字节。然而这对于用户来说不容易查看，所以df和du的输出默认以1024字节为单位。如果你想使用POSIX的512字节为单位，可以设置POSIXLY_CORRECT环境变量。也可以使用-k选项来特别指定使用1024字节块（df和du均支持）。df还有一个-m选项，用于以1MB为单位显示容量，-h则是自动判断和使用对用户来说容易理解的单位。

4.2.11　检查和修复文件系统

Unix文件系统通过一个复杂的数据库机制来提供性能优化。为了让各种文件系统顺畅地工作，内核必须信任加载的文件系统不会出错。如果出错则会导致数据丢失和系统崩溃。

文件系统错误通常是由于用户强行关闭系统导致（例如直接拔掉电源）。此时文件系统在内存中的缓存与磁盘上的数据有可能会有出入，或者系统有可能正在更改文件系统。虽然新的文件系统支持日志功能来防止数据损坏，你仍然要使用恰当的方式来关闭系统。文件系统检查也是保证数据完整的必要措施。

检查文件系统的工具是fsck。对于mkfs来说，fsck对每个Linux支持的文件系统类型都有一个对应版本。例如，当你在扩展文件系统（ext2/ext3/ext4）上运行fsck时，fsck检测到文件系统类型，并且启动e2fsck工具。所以通常你不需要自己指定e2fsck，除非fsck无法识别文件系统类型或者你正在查看e2fsck使用手册。

本节中涉及的信息是针对扩展文件系统和e2fsck。

要在手动交互模式下运行fsck，可以指定设备或者挂载点（/etc/fstab中）作为参数，如：

```
# fsck /dev/sdb1
```

警告　不可以在一个已经挂载的文件系统上使用fsck，因为内核在你执行检查时有可能更新磁盘数据，导致运行时数据不匹配，从而导致系统崩溃和文件损坏。只有一个例外，就是你使用单用户只读模式挂载root分区时，可以在上面使用fsck。

在手动模式下，fsck会输出很多状态信息，如下所示：

```
Pass 1: Checking inode, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
/dev/sdb1: 11/1976 files (0.0% non-contiguous), 265/7891 blocks
```

fsck在手动模式下发现问题会停止，并且问你关于如何修复的一个问题。问题涉及文件系统的内部结构，诸如重新连接inode（它是文件系统的基本组成部分，我们将在4.5节介绍），清除区块等。如果fsck问你是否重新连接inode，说明它发现了一个未命名文件。在重新连接这个文件时，fsck将文件放到lost+found目录中，并使用一个数字作为文件名，因为原来的文件名很可能已经丢失节点，你需要根据文件内容为文件起个新的名字。

如果你仅仅是非正常关闭了系统，通常不需要运行fsck来修复文件系统，因为fsck可能会产生许多大大小小的报错。还好e2fsck有一个选项-p能够自动修复常见的错误，遇到严重错误时则会终止。实际上很多Linux发行版在启动时会运行各自的fsck -p版本。（你可能还见过fsck -a，它的功能是一样的。）

如果你觉得系统出现了一个严重的问题，比如硬件故障或者设备配置错误，这时就需要仔细斟酌如何采取相应措施，因为fsck有可能会帮倒忙。（如果fsck在手动模式下向你提出许多问题，可能意味着系统出现了很严重的问题。）

如果你觉得系统出现了很严重的故障，可以运行fsck -n来检查文件系统，同时不做任何更改。如果问题出在设备配置方面，并且你能够修复（比如分区表中的块数目不正确或者数据线没插稳），那就请在运行fsck之前修复它，否则你有可能丢失很多数据。

如果你认为超级块被损坏了（比如有人在磁盘分区最开始的位置写入了数据），你可以使用mkfs创建的超级块备份来恢复文件系统。你可以运行fsck -b num命令，以使用位于num的块来替换损坏的超级块。

如果你不知道在哪里能找到备份超级块，可以运行mkfs -n来查看备份超级块列表，这不会损失任何数据。（再次强调，你必须使用-n选项，否则真的会损坏文件系统。）

检查ext3和ext4文件系统

一般情况下，你不需要手动检查ext3和ext4文件系统，因为日志保证了数据完整性。然而，你可能想要在ext2模式中挂载一个已损坏的ext3或者ext4文件系统,因为内核无法使用非空日志来挂载ext3和ext4文件系统。（如果你没有正常关闭系统的话，日志里有可能会残留数据。）你可以运行以下e2fsck命令来将ext3和ext4文件系统中的日志数据写入到数据库。

```
# e2fsck –fy /dev/disk_device
```

最坏的情况

面对严重的磁盘故障，你可以有下面的选择。

- 你可以使用dd尝试从磁盘提取出整个文件系统的映像，然后将它转移到另一块磁盘的相同大小的分区中。

- 你可以在只读模式下挂载文件系统，然后再想办法修复。
- 使用`debugfs`。

对于前两个选项，你必须先修复文件系统，然后才能挂载它，除非你愿意手动遍历磁盘数据。你可以使用`fsck -y`来回答所有的`fsck`提示，不过如果不是迫不得已不要使用这个方法，因为它有可能带来更多的问题，还不如你手动处理。

`debugfs`工具能够让你遍历文件系统中的文件，并将它们复制到其他地方。默认情况下，它在只读模式下打开文件系统。如果你要恢复数据的话，最好确保文件的完整性，以免让事情变得更糟。

最坏的情况下，比如磁盘严重损坏并且没有备份，你能做的也只能是向专业的数据修复服务商寻求帮助了。

4.2.12　特殊用途的文件系统

文件系统并不仅仅用于存储在物理媒介上的数据。大多数Unix系统中都有一些作为系统接口来使用的文件系统。也就是说，文件系统不仅仅为在存储设备上存储数据提供服务，还能够用来表示系统信息，如PID和内核诊断信息。这个思路和/dev类似，即使用文件作为I/O接口。/proc出自research Unix的第八版，由Tom J. Killian实现，在Bell实验室（其中有很多最初的Unix设计者）创造Plan 9的时候得以促进。Plan 9是一个科研用操作系统，它将文件系统的抽象程度提升到了一个新的高度（http://plan9.bell-labs.com/sys/doc/9.html）。

Linux中特殊用途的文件系统有以下类型。

- proc：挂载在/proc。proc是进程（process）的缩写。/proc目录中的子目录以系统中的PID命名，子目录中的文件代表的是进程的各种状态。文件/proc/self表示当前进程。Linux系统的proc文件系统包括大量的内核和硬件系统信息，保存在像/proc/cpuinfo这样的文件中。（后来有人提出将进程无关的信息从/proc移至/sys。）
- sysfs：挂载在/sys（在第3章已经介绍过）。
- tmpfs：挂载在/run和其他位置。通过tmpfs，你可以将物理内存和交换空间作为临时存储。例如，你可以将tmpfs挂载到任意位置，使用`size`和`nr_blocks`长选项来设置最大容量。但是请注意不要经常随意地将数据存放到tmpfs，这样你很容易占满系统内存，会导致程序崩溃。（多年来，Sun Microsystems公司使用的tmpfs在长时间运行后会导致一系列问题。）

4.3　交换空间

并不是所有磁盘分区都包含文件系统。系统可以通过使用磁盘空间来扩展内存容量。如果出现内存空间不足的情况，Linux虚拟内存系统会自动将内存中的进程移出至磁盘以及从磁盘移入内存。我们称其为*交换*（swap），因为空闲的进程被移出到磁盘，同时被激活的进程从磁盘移入到内存。用来保存内存页面的磁盘空间我们称为*交换空间*（swap space，或简称swap）。

使用`free`命令可以显示当前交换空间的使用情况：

```
$ free
            total       used       free
--snip--
Swap:      514072     189804     324268
```

4.3.1 使用磁盘分区作为交换空间

通过以下步骤来将整个磁盘分区作为交换空间。

(1) 确保分区为空。

(2) 运行mkswap dev，其中dev是分区设备。该命令在分区上放置一个交换签名。

(3) 运行swapon dev向内核注册。

交换分区创建后，你可以在/etc/fstab文件中创建一个新的交换条目，这样系统在重启之后即可使用该交换空间。以下是一个条目的样例，使用/dev/sda5作为交换分区：

```
/dev/sda5 none swap sw 0 0
```

请记住，现在很多系统使用的是UUID而非原始设备名。

4.3.2 使用文件作为交换空间

如果不想重新分区或者不想新建交换分区的话，你可以使用常规文件作为交换空间，它们的效果是一样的。

具体步骤为创建一个空文件，将其初始化为交换空间，然后将其加入交换池。所用的代码如下例所示：

```
# dd if=/dev/zero of=swap_file bs=1024k count=num_mb
# mkswap swap_file
# swapon swap_file
```

这里swap_file是新交换文件的名称，num_mb是需要的文件大小，以MB为单位。

可以使用swapoff命令来删除交换分区或交换文件。

4.3.3 你需要多大的交换空间

以前Unix系统建议将交换空间大小设置为内存容量的至少两倍。如今内存和磁盘空间都不再是问题，关键看我们怎样使用系统。一方面，海量的磁盘空间让我们可以分配超过两倍内存的交换空间，另一方面，内存大到我们根本不需要交换空间。

“两倍内存容量”这一规则对于多个用户系统来说就显得过时了。由于并不是所有用户都处于活跃状态，最好能够将非活跃用户的内存交换给那些有需要的活跃用户。

对于单用户系统来说，此规则仍适用。如果你在运行多个进程，可以交换不活跃进程，甚至交换活跃进程中的那些不活跃部分都没问题。如果许多活跃进程同时都需要内存，因而必须频繁地使用交换空间，这时你会碰到非常严重的性能问题，因为磁盘I/O速度相对较慢。唯一的解决

办法就是增加更多内存，终止一些进程，或者吐槽一下。

有时候Linux内核可能会为了获得磁盘缓冲而交换出一个进程。为了防止这种情况，有些系统管理员将系统设置为不允许交换空间。例如，高性能网络服务器需要尽可能避免磁盘存取和交换空间。

注解　如果系统耗尽了所有的物理内存和交换空间，Linux内核会调用out-of-memory（OOM）来终止一个进程以获得一些内存空间。你应该不希望你的桌面应用程序被这样终止。另一方面，高性能服务器有复杂的监控和负载平衡系统来保证内存不会被完全耗尽。

我们将在第8章详细介绍内存系统。

4.4　前瞻：磁盘和用户空间

对于Unix系统上那些磁盘相关的组件，我们很难界定用户空间和内核空间的边界。内核处理设备上的基本块I/O，用户空间工具可以通过设备文件来使用块I/O。然而，用户空间通常只使用块I/O做一些初始化操作，比如分区、创建文件系统和创建交换分区。一般情况下，用户空间只在块I/O的基础上使用内核提供的文件系统支持。类似地，内核在虚拟内存系统中处理交换空间时也负责处理大部分繁琐的细节。

本章后面的内容将简要介绍Linux文件系统的内部结构。这些内容主要针对高级用户，不影响普通读者的阅读，你可以直接跳到下一章学习Linux的启动。

4.5　深入传统文件系统

传统的Unix文件系统有两个基础组件：一个用来存储数据的数据块池和一个用来管理数据池的数据库系统。这个数据库是inode数据结构的核心。inode是一组描述文件的数据，包括文件类型、权限，以及最重要的一点，即文件数据所在的数据池。inode在inode表中以数字的形式表示。

文件名和目录也是通过inode来实现的。目录inode包含一个文件名列表以及对应的指向其他inode的链接。

为了方便举例，我们来创建一个新的文件系统，挂载它，并切换到挂载点目录。然后加入一些文件和目录（你可以在一个闪存盘上来做实验）：

```
$ mkdir dir_1
$ mkdir dir_2
$ echo a > dir_1/file_1
$ echo b > dir_1/file_2
$ echo c > dir_1/file_3
$ echo d > dir_2/file_4
$ ln dir_1/file_3 dir_2/file_5
```

这里我们创建了一个硬链接dir_2/file_5，指向dir_1/file_3，它们实际上代表的是同一个文件（稍后详述）。

从用户角度而言，该文件系统的目录结构如图4-4所示。而图4-5更为复杂，显示的是真实的文件系统结构。

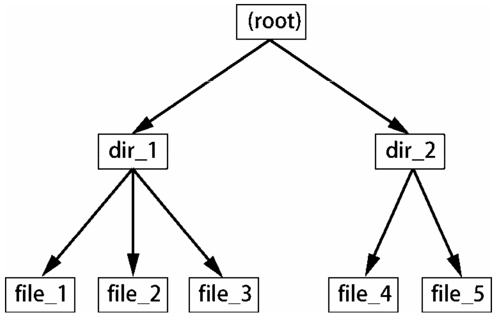


图4-4　用户眼中的文件系统

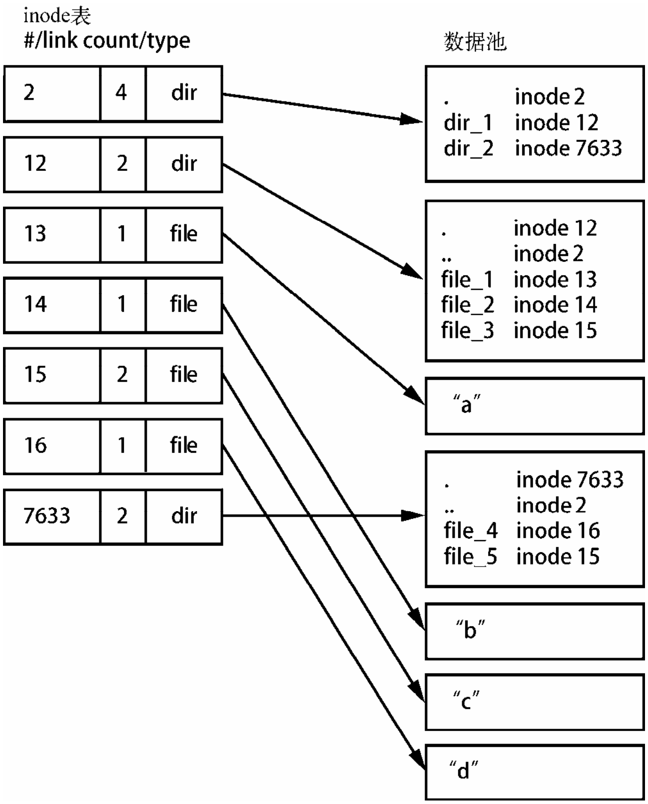


图4-5　图4-4对应的inode文件系统结构

我们如何来理解这个图呢？对ext2/3/4文件系统来说，编号为2的inode是root inode，它是一个目录inode（即dir）。如果跟随箭头到数据池，我们可以看到root目录的内容：dir_1和dir_2两个条目分别对应inode 12和7633。我们也可以回到inode表查看这两个inode的详细内容。

内核采取以下步骤来对dir_1/file_2做检查。

(1) 检查路径部分，即目录dir_1和后面的file_2。

(2) 通过root inode找到它的目录信息。

(3) 在inode 2的目录信息中找到dir_1，它指向inode 12。

(4) 在inode表中查找inode 12，验证它是一个目录。

(5) 找到inode 12的目录信息（在数据池中）。

(6) 在inode 12的目录信息中找到file_2，它指向inode 14。

(7) 在目录表中查找inode 14，它是一个文件inode。

至此，内核了解了该文件的属性，可以通过inode 14的数据链接打开它了。通过这种方式，inode指向目录数据结构，目录数据结构也指向inode，这样你可以根据自己的习惯创建文件系统结构。另外请注意目录inode中包含了.和..两个条目（除root目录以外），让你能够轻松地在目录结构中浏览。

4.5.1　查看inode细节

我们可以使用命令`ls -i`来查看目录的inode编号。例如上例中的目录inode编号如下所示（可以使用`stat`命令来查看更详细的信息）：

```
$ ls -i
  12 dir_1 7633 dir_2
```

你可能在`ls -l`命令的结果中见过但忽略了**链接计数**（link count）这个信息，图4-5中文件的链接计数是多少呢，特别是硬链接file_5？链接计数是指向同一个inode的所有目录条目的总数。大多数文件的链接计数是1，因为它们大多在目录条目里只出现一次。这不奇怪，因为通常当你创建一个新文件的时候，你只为其创建一个新的目录条目和一个新的inode。然而inode15出现了两次：一次是dir_1/file_3，另一次是dir_2/file_5。硬链接是手动创建的、指向一个已有的inode的目录条目。使用`ln`命令（不带`-s`选项）可以创建新链接。

这就是为什么我们有时候将删除文件称为取消链接。如果你运行`rm dir_1/file_2`，内核会在inode 12的目录条目中搜索名为file_2的条目。当发现file_2对应inode 14的时候，内核删除目录条目，同时将inode 14的链接计数减1。这导致inode 14的链接计数为0，内核发现该inode没有任何链接的时候，会将其和与之相关的所有数据删除。

但是如果你运行`rm dir_1/file_3`，inode 15的链接计数会由2变为1（dir_2/file_5仍然与之链接），这时内核不会删除此inode。

链接计数对于目录来说也是同理。inode 12的链接计数为2，一个是目录条目中的dir_1（inode 2），另一个是它自己的目录条目中的自引用（.）。如果你创建一个新目录dir_1/dir_3，inode 12的

链接计数会变为3，因为新目录包含其上级目录（..）条目，而这个条目又链接到inode 12。类似inode 12指向其上级目录inode 2。

有一种情况比较特殊，root目录的inode 2的链接计数为4。而图4-5中只显示了3个目录条目链接。另外一个其实是文件系统的超级块，它知道如何找到root inode。

你完全可以自己做一些尝试，使用`ls -i`创建文件系统，使用`stat`来遍历，这些操作都很安全。你也不需要使用root权限（除非你需要挂载和创建新的文件系统）。

有一个地方我们还没有讲到，就是在为新文件分配数据池块的时候，文件系统如何知道哪些块可用，哪些已被占用？方法之一是使用块位图（block bitmap）来管理块信息。文件系统保留了一些字节空间，每一位代表一个数据池中的一个块。0代表块可用，1代表块已经被占用，释放和分配块就变成了0和1之间的切换。

当inode表中的数据和块分配数据不匹配，或者由于你没有正确关闭系统导致链接数目不正确，文件系统就会出错。所以你在检查文件系统的时候，如4.1.11节介绍的那样，`fsck`会遍历inode表和目录结构来生成链接数目和块分配信息，并且会根据磁盘上的文件系统来检查新数据，如果发现数据不匹配的情况，`fsck`会修复链接数错误以及inode和其他一些目录结构数据错误。大部分`fsck`程序会将新创建的文件放到lost+found目录。

4.5.2 在用户空间中使用文件系统

在用户空间中使用文件和目录时，你不需要太关心底层实现的细节。你只需要能够通过内核系统调用来访问文件和目录的内容。其实你也能够看到一些看似超出用户空间范围的文件系统信息，特别是`stat()`这个系统调用能够告诉你inode数目和链接计数。

除非你需要维护文件系统，否则你不需要知道inode数目和链接计数。用户模式中的程序之所以能够访问这些信息，主要是因为一些向后兼容的考虑。并且也不是所有Linux的文件系统都提供这些信息。VFS接口层能够确保系统调用总是返回inode数目和链接计数，不过这些数据可能没有太大意义。

在传统文件系统上你有可能无法执行一些传统的Unix文件系统的操作。比如，你无法使用`ln`命令在VFAT文件系统上创建硬链接，因为其目录条目数据结构根本不同。

幸运的是Unix/Linux提供给用户空间的系统调用为用户访问文件提供了足够的便利，用户不需要关心任何底层的细节。文件命名很灵活，支持大小写混合，并且能很容易地支持其他文件系统结构。

请记住，内核只是系统调用的通道，而不会包含对某个特定的文件系统的支持。

4.5.3 文件系统的演进

你已经看到了，就算一个很简单的文件系统也包括各种各样不同的组件，需要去维护。同时随着新需求、新技术和存储容量的不断发展，用户对文件系统的要求也越来越高。如今，性能、数据完整性、安全性等方面的需求大大超出了老式文件系统的功能范围，因而文件系统方面的技

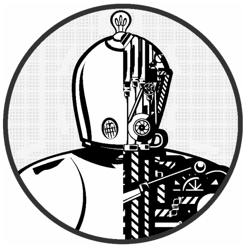
术也在日新月异地发展。一个例子就是我们在4.2.1节中提到的Btrfs。

文件系统演进的另一个例子是新的文件系统使用不同的数据结构来表示文件和目录。它们使用数据块的方式各不相同，而不是使用本章介绍的inode。此外，针对SSD优化的文件系统也在不断演进。但无论怎么变化，归根结底它们的最终目的都是一样的。

第 5 章

Linux内核的启动

5



目前为止，我们介绍了Linux系统的物理结构和逻辑结构、内核以及进程。本章我们将讨论内核的启动，即从内核载入内存到启动第一个用户进程的过程。

下面是简化后的整个启动过程。

(1) BIOS或者启动固件加载并运行引导装载程序。

(2) 引导装载程序在磁盘上找到内核映像，将其载入内存并启动。

(3) 内核初始化设备及其驱动程序。

(4) 内核挂载root文件系统。

(5) 内核使用PID 1来运行一个叫init的程序，用户空间在此时开始启动。

(6) init启动其他的系统进程。

(7) init还会启动一个进程，通常发生在整个过程的尾声，负责用户登录。

本章涉及前4个步骤，主要介绍内核和引导装载程序。在第6章我们会介绍用户空间启动。

理解启动过程对将来修复启动相关的问题会大有帮助，也有助于了解整个Linux系统。然而，你很难从Linux系统的默认启动过程中分辨出最前面的几个步骤，通常只有在整个过程完成，你登录系统后才有机会看到。

5.1　启动消息

传统的Unix系统在启动时会显示很多系统信息，方便你查看启动过程。这些消息一开始来自内核，然后是进程和init执行的初始化程序。然而，这些消息格式不是那么清晰和一致，有时甚至不是那么有用。现在大部分Linux发行版都使用启动屏幕、屏幕填充色和启动选项菜单将它们遮盖住。此外，硬件性能的提升也让启动过程更快，这些消息显示得也更快，快到让人难以捕捉。

有两种方法可以查看内核启动信息和运行时的诊断信息。

- 查看内核系统日志文件。这些文件通常存放在/var/log/kern.log中，取决于你的系统配置，也可能和其他系统日志一起存放在/var/log/messages或者别的地方。
- 使用dmesg命令，不过记得将结果输出到less，因为信息量会比较大。dmesg命令使用内核环缓冲区，它的容量有限，不过大多数较新的内核能够有足够的空间来容纳足够长的启动日志。

以下是一个dmesg的示例。

```
$ dmesg
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 3.2.0-67-generic-pae (buildd@toyol) (gcc version 4.
6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5) ) #101-Ubuntu SMP Tue Jul 15 18:04:54 UTC 2014
 (Ubuntu 3.2.0-67.101-generic-pae 3.2.60)
[    0.000000] KERNEL supported cpus:
--snip--
[    2.986148] sr0: scsi3-mmc drive: 24x/8x writer dvd-ram cd/rw xa/form2 cdda tray
[    2.986153] cdrom: Uniform CD-ROM driver Revision: 3.20
[    2.986316] sr 1:0:0:0: Attached scsi CD-ROM sr0
[    2.986416] sr 1:0:0:0: Attached scsi generic sg1 type 5
[    3.007862]  sda: sda1 sda2 < sda5 >
[    3.008658] sd 0:0:0:0: [sda] Attached SCSI disk
--snip--
```

内核启动后，用户空间启动程序会产生消息。查看这些消息不是很方便，因为它们分布在很多不同的地方。启动脚本通常会将消息显示到屏幕上，启动完成后就从屏幕上消失了。但这也不是大问题，因为每个脚本都将消息写入它们各自的日志。有一些版本的init，比如Upstart和systemd，可以获得那些显示到屏幕的启动和运行时消息。

5.2　内核初始化和启动选项

在启动时，Linux内核的初始化过程如下：

(1) 检查CPU；

(2) 检查内存；

(3) 检测设备总线；

(4) 检测设备；

(5) 设置附加内核子系统（如网络等）；

(6) 挂载root目录；

(7) 启动用户空间。

前面几个步骤很好理解，设备相关的步骤则涉及一些依赖性问题。例如磁盘设备驱动程序可能需要依赖于总线和SCSI子系统的支持。

在后面的几个步骤中，内核必须在初始化前挂载root文件系统。你可以忽略大部分细节，除

了一点，就是一些组件并不是主内核的一部分，它们会以可加载内核模块的方式启动。在一些系统中，你可能需要在挂载root文件系统之前加载这些内核模块。详细内容我们将在6.8节介绍。

直到本书写成时，内核在启动第一个用户进程时不会显示任何消息。然而下面的这些与内存管理相关的消息能够为我们提供一些信息，内核将自己的内存空间保护起来以防止用户空间进程使用。下面这些信息预示着用户空间即将启动：

```
Freeing unused kernel memory: 740k freed
Write protecting the kernel text: 5820k
Write protecting the kernel read-only data: 2376k
NX-protecting the kernel data: 4420k
```

这时你还可以看到root文件系统的挂载信息。

注解 本章下面的内容涉及内核启动的细节，你可以跳到第6章学习用户空间启动以及内核初始化第一个用户进程等内容。

5.3 内核参数

运行Linux内核的时候，引导装载程序会向内核传递一系列文本形式的内核参数来设定内核的启动方式。这些参数设定了很多不同的行为方式，比如内核显示的诊断信息的多少、设备驱动程序相关参数等。

你可以通过/proc/cmdline文件来查看系统启动时使用的内核参数：

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-3.2.0-67-generic-pae root=UUID=70ccd6e7-6ae6-44f6-
  812c-51aab8036d29 ro quiet splash vt.handoff=7
```

这些参数有的是一个单词的长度，诸如`ro`、`quiet`，有的是`key=value`这样的配对（例如`vt.handoff=7`）。它们中大部分都无关紧要，如`splash`标记的意思是显示一个闪屏（splash screen）。但`root`参数很重要，它是root文件系统存放的位置，如果没有这个参数，内核将无法完成初始化工作，从而也就无法启动用户空间。

root文件系统参数值是一个设备文件，例如：

```
root=/dev/sda1
```

然而现在的桌面系统中经常使用UUID（见4.2.4 节）：

```
root=UUID=70ccd6e7-6ae6-44f6-812c-51aab8036d29
```

参数`ro`告诉内核在用户空间启动时以只读模式挂载root文件系统。（使用只读模式能够让

fsck安全地对root文件系统做检查，之后启动进程重新以可读写模式来挂载root文件系统。）

如果遇到无法识别的参数，Linux内核会将其保存，稍后在启动用户空间时传递给init。如果你加入参数-s，内核会将其传递给init，让其以单用户模式启动。

下面我们介绍引导装载程序是如何启动内核的。

5.4　引导装载程序

在启动过程的最开始，引导装载程序启动内核，然后内核和init启动。引导装载程序的工作看似很简单：将内核加载到内存，然后使用一系列内核参数启动内核。但是关于引导装载程序，需要弄清楚下面两个问题。

- 内核在哪里？
- 内核启动时需要传递哪些参数？

答案是：内核及其参数（通常是）在root文件系统中。因为内核此时还没有开始运行，无法遍历文件系统，所以内核参数需要被放到一个容易存取的地方。并且此时用于访问磁盘的内核设备驱动还没有准备好，这听起来有点像“鸡生蛋，蛋生鸡”。

我们先看一下驱动程序。在个人电脑上，引导装载程序使用*基本输入输出系统*（以下简称BIOS）或者*统一可扩展固件接口*（Unified Extensible Firmware Interface，以下简称UEFI）来访问磁盘。几乎所有的磁盘设备都有固件系统供BIOS通过*线性块寻址*（Linear Block Addressing）来访问硬件。虽然性能不怎么样，但是这种方式可以访问磁盘的任意位置。引导装载程序往往是唯一使用BIOS访问磁盘的程序。内核使用的是它自己的高性能驱动程序。

大多数现在的引导装载程序都能够读取分区表，内建以只读模式访问文件系统的功能，因此它们能够查找和读取文件。这使得动态配置和完善引导装载程序变得非常简单。并不是所有的Linux引导装载程序都有这些功能，配置引导装载程序因而变得困难得多。

5.4.1　引导装载程序任务

Linux引导装载程序的核心功能如下：

- 从多个内核中选择一个使用；
- 从多个内核参数集中选择一个使用；
- 允许用户手动更改内核映像名和参数（例如使用单用户模式）；
- 支持其他操作系统的启动。

自Linux内核问世以来，引导装载程序的功能得到了极大的增强，加入了历史记录和菜单界面，不过最基本的需求仍然是能够灵活选择内核映像和参数。有趣的是某些方面的需求逐渐消失了。例如，现在你可以从USB设备上执行紧急启动和恢复，因而你可能不再需要手工设置内核参数和进入单用户模式。不过因为现在引导装载程序强大的功能，更改内核参数、创建定制内核这些事情也变得容易得多。

5.4.2 引导装载程序概述

以下是一些常见的引导装载程序，按照普及的顺序排列。

- GRUB：近乎于Linux系统标准。
- LILO：最早期的Linux引导装载程序之一，是UEFI的一个版本。
- SYSLINUX：能够在很多不同的文件系统上配置和启动。
- LOADLIN：能够从MS-DOS上启动内核。
- efilinux：UEFI引导装载程序的一种，作为其他UEFI引导装载程序的模块和引用。
- coreboot（以前又叫作LinuxBIOS）：PC BIOS的高性能替代品，并且能够包含内核。
- Linux Kernel EFISTUB：能够从EFI/UEFI系统分区（ESP）加载内核的一个内核插件。

本书只涉及GRUB。其他引导装载程序也有某些优于GRUB的地方，比如更容易配置或更快速。

要设置内核名称和参数，你首先需要了解如何进入启动提示符。因为很多Linux系统定制了引导装载程序，使得我们有时很难找到进入启动提示符的方法。 下一节我们会介绍如何进入启动提示符来设置内核名称和参数，然后你将了解如何设置和安装引导装载程序。

5.5 GRUB 简介

GRUB意指大一统引导装载程序（Grand Unified Boot Loader）。本节我们将介绍GRUB 2。GRUB有一个较老的版本叫作GRUB Legacy，现在逐渐被淘汰了。

GRUB最重要的一个功能是对内核映像和配置的选择更为简便。我们可以通过查看菜单来了解GRUB。GRUB界面易于操作，不过Linux的发行版尽可能将引导装载程序隐藏起来，你可能没机会看到。

你可以在BIOS或者固件启动屏幕出现时按住SHIFT来打开GRUB菜单。图5-1中是GRUB菜单，可以按ESC来暂时取消自动启动计时。

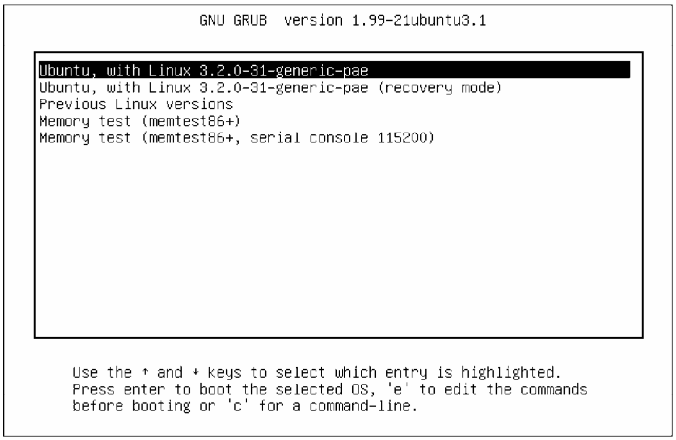


图5-1 GRUB菜单

可以通过以下步骤来查看引导装载程序。

(1) 打开或者重启Linux。

(2) 在BIOS/固件自检时或者启动屏幕显示时，按住SHIFT显示GRUB菜单。

(3) 按E键查看引导装载程序命令的默认启动选项，如图5-2所示。

```
GNU GRUB  version 1.99-21ubuntu3.1

setparams 'Ubuntu, with Linux 3.2.0-31-generic-pae'

recordfail
gfxmode $linux_gfx_mode
insmod gzio
insmod part_msdos
insmod ext2
set root='(hd0,msdos1)'
search --no-floppy --fs-uuid --set=root 4898e145-b064-45bd-b7b4-7326\
b00273b7
linux /boot/vmlinuz-3.2.0-31-generic-pae root=UUID=4898e145-b064-45b\
d-b7b4-7326b00273b7 ro   quiet splash $vt_handoff
initrd /boot/initrd.img-3.2.0-31-generic-pae

Minimum Emacs-like screen editing is supported. TAB lists
completions. Press Ctrl-x or F10 to boot, Ctrl-c or F2 for
a command-line or ESC to discard edits and return to the GRUB
menu.
```

图5-2　GRUB配置编辑器

如图所示，root文件系统被设置为一个UUID，内核映像是`/boot/vmlinuz-3.2.0-31-generic-pae`，内核参数包括`ro`、`quiet`和`splash`。初始RAM文件系统是`/boot/initrd.img-3.2.0-31-generic-pae`。如果你从来没有见过这些配置信息，可能会觉得比较糊涂。你可能会问为什么会有这么多地方涉及root？它们有什么区别？为什么这里会出现insmod？Linux内核不是通常由`udevd`运行吗？

我一说你就明白了，因为GRUB仅仅是*启动*内核，而不是*使用*它。你看到的这些配置信息都由GRUB的内部命令构成，GRUB自身另成一个体系。

产生混淆的原因是GRUB借用了很多其他地方的术语。GRUB有自己的“内核”和`insmod`命令来动态加载GRUB模块，这和Linux内核没有任何关系。很多GRUB命令和Unix shell命令很类似，GRUB也有一个`ls`命令来显示文件列表。

不过最让人糊涂的地方还是root这个词。为了表达得更清楚，我们只需要遵循一个简单的法则，就是：只有`root`*内核*参数是指root文件系统。

在GRUB配置中，`root`这个内核参数在`linux`命令中的映像文件名后面。其他配置信息中出现root的地方指的都是GRUB root，它只针对GRUB，是GRUB查找内核和RAM文件系统映像时使用的文件系统。

在图5-2中，GRUB root一开始被设置为GRUB相关设备（hd0,msdos1）。在随后的命令中，

GRUB查找一个特定分区的UUID，如果找到的话就将该分区设置为GRUB root。

总而言之，`linux`命令的第一个参数（`/boot/vmlinuz-...`）是Linux内核映像文件的位置。GRUB从GRUB root上加载此文件。类似地，`initrd`命令指定初始的RAM文件系统文件。

你可以在GRUB内配置这些信息，通常启动错误可以用这种方式来暂时性地修复。如果要永久性地修复启动问题，你需要更改配置信息（见5.5.2节），不过目前让我们先深入GRUB内部，看看其命令行界面。

5.5.1 使用GRUB命令行浏览设备和分区

5

如图5-2所示，GRUB有自己的设备寻址方式。例如，系统检测到的第一个硬盘是hd0，然后是hd1，以此类推。然而设备分配会有变化。还好GRUB能够在所有的分区中通过UUID来查找得内核所在的分区，就像你刚才在`search`命令中看到的那样。

设备列表

想要了解GRUB如何显示系统中的设备，可以在启动菜单或者配置编辑器中按C键进入GRUB命令行。你将看到以下提示符：

```
grub>
```

这里你可以运行任何你在配置信息中看到的命令。我们可以从`ls`命令开始，不带参数，命令的输出结果是GRUB能够识别的所有设备的列表：

```
grub> ls
(hd0) (hd0,msdos1) (hd0,msdos5)
```

本例中有一个名为hd0的主磁盘设备和分区（hd0,msdos1）及（hd0,msdos5）。前缀msdos表示磁盘包含MBR分区表。如果包含的是GPT分区表，则前缀就是gpt。（还可能会有第三个标识符来表示更多可能的组合，如分区包含BSD磁盘标签映射，不过你通常不需要太关注，除非你在一台机器上运行多个操作系统。）

使用`ls -l`命令可以查看更详细的信息。此命令能显示磁盘上所有分区的UUID，所以非常有用，例如：

```
grub> ls -l
Device hd0: Not a known filesystem - Total size 426743808 sectors
        Partition hd0,msdos1: Filesystem type ext2 – Last modification time
          2015-09-18 20:45:00 Friday, UUID 4898e145-b064-45bd-b7b4-7326b00273b7 -
Partition start at 2048 - Total size 424644608 sectors
        Partition hd0,msdos5: Not a known filesystem - Partition start at
          424648704 - Total size 2093056 sectors
```

如上所示，磁盘在第一个MBR分区上有一个Linux ext2/3/4文件系统，在分区5上有一个Linux交换区签名，这样的配置很常见。（从输出中可以看到(hd0,msdos5)是交换分区。）

文件导航

现在我们来看看GRUB的文件系统导航。你可以使用echo命令来查看GRUB root文件系统（这是GRUB寻找内核的地方）：

```
grub> echo $root
hd0,msdos1
```

可以在GRUB的ls命令中的分区后面加上/来显示root文件系统下的文件和目录：

```
grub> ls (hd0,msdos1)/
```

不过要记住键入分区名是件麻烦事，为了节省时间，你可以使用root变量：

```
grub> ls ($root)/
```

输出结果是该分区上的文件系统中的目录和文件名列表，诸如etc/、bin/和dev/。你需要了解的是，这个命令和GRUB ls的功能完全不同，后者列出设备、分区表盒文件系统头信息，而这个命令显示的是文件系统的内容。

使用类似的方法，你可以进一步查看分区上的文件和目录的内容。例如，如果要查看boot目录的内容，可以使用：

```
grub> ls ($root)/boot
```

注解 你可以使用上下键来查看命令历史记录，使用左右键来编辑当前命令行。也可以使用标准行命令如：CTRL-N、CTRL-P等。

你也可以使用set命令查看所有已经设置的GRUB变量：

```
grub> set
?=0
color_highlight=black/white
color_normal=white/black
--snip--
prefix=(hd0,msdos1)/boot/grub
root=hd0,msdos1
```

这些变量当中，$prefix是很重要的一个，GRUB使用它指定的文件系统和目录来寻找配置和辅助支持信息。我们将在下一节详细介绍。

使用GRUB命令行界面完成工作后，你可以键入boot命令来启动系统，或者按ESC键回到GRUB菜单来启动系统。整个系统启动完毕准备就绪之后，让我们来看看GRUB配置信息。

5.5.2 GRUB配置信息

GRUB配置目录包含核心配置文件（grub.cfg）和一些可加载模块，以.mod为后缀。（随着GRUB版本的演进，这些模块被逐渐移到像i386-pc这样的子目录中。）配置目录通常是/boot/grub或者/boot/grub2。我们不直接编辑grub.cfg文件，而是使用`grub-mkconfig`命令（或者Fedora上的`grub2-mkconfig`）。

回顾grub.cfg

首先我们看一下GRUB是如何通过grub.cfg文件来初始化菜单和内核选项的。你会看到grub.cfg文件中包含GRUB命令，通常是从一系列的初始化步骤开始，然后是一系列的菜单条目，针对不同的内核和启动配置。初始化过程并不复杂，它就是一系列的函数定义和视频设置命令，如下所示：


```
if loadfont /usr/share/grub/unicode.pf2 ; then
  set gfxmode=auto
  load_video
  insmod gfxterm
  --snip--
```

稍后在文件中你还会看到启动配置信息，它们以`menuentry`命令开始。通过上节的介绍，你应该能够理解以下这些内容：

```
menuentry 'Ubuntu, with Linux 3.2.0-34-generic-pae' --class ubuntu -- class gnu-linux --class gnu
--class os {
        recordfail
        gfxmode $linux_gfx_mode
        insmod gzio
        insmod part_msdos
        insmod ext2
        set root='(hd0,msdos1)'
        search --no-floppy --fs-uuid --set=root 70ccd6e7-6ae6-44f6-812c-51aab8036d29
        linux   /boot/vmlinuz-3.2.0-34-generic-pae root=UUID=70ccd6e7-6ae6-44f6-812c-51aab8036d29
          ro   quiet splash $vt_handoff
        initrd  /boot/initrd.img-3.2.0-34-generic-pae
}
```

请留意`submenu`命令。如果你的grub.cfg文件包含一系列`menuentry`命令，它们中大多数可能被包含在`submenu`命令中，这是针对较老版本的内核设计的，以免GRUB菜单过于臃肿。

生成新的配置文件

如果想要更改GRUB配置，我们不会直接编辑grub.cfg文件，因为它是由系统自动生成和更新的。你可以将新的配置信息放到其他地方，然后运行`grub-mkconfig`来生成新的配置文件。

在grub.cfg文件的最开头应该有一行注释，如下：

```
### BEGIN /etc/grub.d/00_header ###
```

你会发现，/etc/grub.d中的文件都是shell脚本，它们各自生成grub.cfg文件的某个部分。grub-mkconfig命令本身也是一个shell脚本文件，负责运行/etc/grub.d中的所有脚本文件。

你可以使用root账号来自己试一试。（不用担心当前的配置信息会被覆盖，该命令只会将配置信息输出到标准输出，即屏幕。）

```
# grub-mkconfig
```

如果你想要在GRUB配置中加入新的菜单条目和其他命令，简单来说，可以在GRUB配置目录中创建一个新的.cfg文件来存放你的内容，如/boot/grub/custom.cfg。

如果涉及细节就会复杂一些了。/etc/grub.d配置目录为你提供了两个选项：40_custom和41_custom。前者是一个脚本文件，你可以编辑，但是系统升级有可能会将你的更改清除掉，所以这个选项不太保险。41_custom脚本更简单一些，它是一组命令，用来在GRUB启动的时候加载custom.cfg文件。（请注意，如果使用该方法，你的配置更改只有在配置文件生成后才会起作用。）

它们并不是唯一定制配置的方法。在一些Linux发行版中，你有可能在/etc/grub.d目录中看到其他的选项。例如：Ubuntu在配置中加入了内存测试启动选项memtest86+。

你可以使用grub-mkconfig命令加-o选项将新生成的文件写到GRUB目录，如下所示：

```
# grub-mkconfig -o /boot/grub/grub.cfg
```

如果你使用的是Ubuntu，可以使用install-grub。在进行这类操作的时候，请注意将旧的配置文件进行备份，以及确保目录路径正确。

现在让我们看看更多关于GRUB和引导装载程序的技术细节，如果你觉得已经了解得足够多，可以直接跳到第6章。

5.5.3　安装GRUB

通常你不用太关注GRUB的安装，Linux系统会自行完成。不过如果你想复制或恢复可启动磁盘，或者自定义启动顺序，可能就需要自己安装GRUB。

在继续以下内容之前，你可以先阅读5.8.3节了解系统如何启动，并在MBR和EFI之间做出选择。接着，你将编译GRUB可执行代码集，并且决定GRUB目录的位置（默认是/boot/grub）。如果Linux系统已经提供了GRUB软件包，你就不必自己动手，否则可以参考第16章来学习如何从源码编译可执行代码。你需要确保目标正确无误。这和MBR以及UEFI启动不同（32位EFI和64位EFI也不同）。

在系统上安装GRUB

安装引导装载程序需要你或者安装程序决定以下几方面。

- 你的运行系统的目标GRUB目录。通常是/boot/grub，不过如果你在不同的系统和磁盘上安装的话，目录位置可能会不一样。
- GRUB目标磁盘对应的设备。

- UEFI启动分区的挂载点，如果是UEFI启动的话。

请注意，GRUB是一个模块化系统，但为了加载模块，它必须要能够读取GRUB目录所在的文件系统。你的任务就是构建一个GRUB系统，它能够读取文件系统，加载配置文件（grub.cfg）以及其他需要的模块。在Linux上，这通常指的是使用预加载的ext2.mod模块来编译一个GRUB系统。编译完成后，你需要将其放到磁盘上的可启动区域，将其他需要用到的文件放到/book/grub目录中。

所幸的是，GRUB自带一个叫作grub-install的工具（注意不要和Ubuntu的install-grub混淆），它负责大部分的GRUB文件安装和配置工作。例如，如果你想在/dev/sda上安装GRUB目录/boot/grub，可以使用以下命令（在MBR上）：


```
# grub-install /dev/sda
```

警告 GRUB安装不正确可能会让系统的启动顺序失效，所以请小心操作。最好是了解一下如何使用dd来备份MBR，并且备份其他已经安装了的GRUB目录，最后确保你有一个应急启动计划。

在外部存储设备上安装GRUB

在当前系统之外安装GRUB，你必须在目标设备上手动指定GRUB目录。例如，你的目标设备是/dev/sdc，其root启动文件系统（root/boot）挂载到系统中的/mnt。这表示当你安装GRUB时，你的系统将会在/mnt/boot/grub中找到GRUB文件。你需要在运行grub-install时指定那些文件的位置：

```
# grub-install --boot-directory=/mnt/boot /dev/sdc
```

使用UEFI安装GRUB

使用UEFI安装应该要简单些。你需要做的就是将引导装载程序拷贝到指定的地方。但是你还需要使用efibootmgr命令向固件注册引导装载程序。grub-install命令会运行这个命令，如果它存在的话。理论上说你要做的就是：

```
# grub-install --efi-directory=efi_dir --bootloader-id=name
```

这里的efi_dir是UEFI目录在你的系统中的位置（通常是/boot/efi/efi，因为UEFI分区通常是挂载到/boot/efi上）。name是引导装载程序的标识符，我们将在5.8.2节介绍。

遗憾的是，在安装UEFI引导装载程序时会出现很多问题。举个例子，如果你在为另一个系统安装磁盘，你需要知道如何向系统的固件注册引导装载程序。此外，可移动媒体的安装程序也不尽相同。

不过最大的问题还是在于UEFI安全启动。

5.6　UEFI 安全启动的问题

最新出现的Linux安装问题之一是安全启动。这个特性在UEFI中打开时，需要引导装载程序被一个信任的机构进行电子签名。微软要求Windows 8供应商使用安全启动。如果你安装了一个没有被签名的引导装载程序（目前大多数Linux系统是这样），它将无法启动。

如果你对Windows不感兴趣，可以在EFI设置中关闭安全启动。然而对于双系统计算机来说这可能不是一个好选择。因此Linux系统提供了经过签名的引导装载程序。一些解决方案只不过是GRUB的一个包装，一些则提供完全的、经过签名的加载过程（从引导装载程序一直到内核），还有一些则是全新的引导装载程序（有一些是基于efilinux）。

5.7　链式加载其他操作系统

UEFI使得加载其他操作系统相对容易，因为你可以在EFI分区上安装多个引导装载程序。然而，较老的MBR不支持这个功能，UEFI也不支持，你还是需要一个单独的分区以及MBR引导装载程序。你可以使用*链式加载*（chainload）来让GRUB加载和运行指定分区上的不同引导装载程序。

要想使用链式加载，你需要在GRUB配置中新建一个菜单条目（使用5.5.2节介绍的方法）。下面是一个在磁盘的第三个分区上启动Windows的例子。

```
menuentry "Windows" {
        insmod chain
        insmod ntfs
        set root=(hd0,3)
        chainloader +1
}
```

选项`+1`告诉`chainloader`加载分区上的第一个扇区中的内容。你还可以使用下面的例子加载io.sys MS-DOS加载程序：

```
menuentry "DOS" {
        insmod chain
        insmod fat
        set root=(hd0,3)
        chainloader /io.sys
}
```

5.8　引导装载程序细节

现在让我们来看一看引导装载程序的细节。如果你对此没有兴趣，可以跳到下一章。

要理解像GRUB这样的引导装载程序的工作原理，首先需要看看你打开计算机时都发生了什么事情。传统计算机启动机制纷繁复杂，有两个主要机制：MBR和UEFI。

5.8.1 MBR启动

除了我们在4.1节中介绍的分区信息之外，主引导记录（Master Boot Record，以下简称MBR）还有一个441字节大小的区域，BIOS在开机自检（Power-On Self-Test，以下简称POST）之后加载其中的内容。然而因为空间太小，无法容纳引导装载程序，所以需要额外的空间，从而就引入了多场景引导装载程序（multi-stage boot loader）。一开始MBR加载引导装载程序的其余部分，通常是在MBR和第一个分区之间。

当然，这样做并不安全，因为里面的代码任何人都可以修改，不过大部分的引导装载程序使用的是这个方式，包括大部分GRUB。另外，这个方式对于GPT分区的磁盘使用BIOS启动不适用，因为GP表信息存放在MBR之后的区域。（为了向后兼容，GPT和传统的MBR分开存储。）

GPT的一种临时解决方案是创建一个小分区，称为BIOS启动分区，即使用一个特别的UUID作为存放完整启动加载代码的地方。但是GPT通常和UEFI一起使用，而不是BIOS，由此引出了UEFI启动方式。

5.8.2 UEFI启动

计算机制造商和软件公司意识到传统的BIOS有很多限制，所以他们决定开发一个替代品，这就是可扩展固件接口（Extensible Firmware Interface，以下简称EFI）。EFI的普及花了一些时间，但现在应用很普遍。目前的标准是统一可扩展固件接口（Unified EFI，以下简称UEFI），包括了诸如内置命令行界面、读取分区表和浏览文件系统等特性。GPT分区方式也是UEFI标准的一部分。

UEFI系统的启动过程非常不同，但大部分都很容易理解。它不是使用存放在文件系统之外的可执行启动代码，而是使用一种特殊的文件系统叫作EFI系统分区（EFI System Partition，以下简称ESP），其中包含一个名为efi的目录。每个引导装载程序有自己的标识符和一个对应的子目录，如efi/microsoft、efi/apple或efi/grub。启动加载文件后缀为.efi，和其他支持文件一起存放在这些目录中。

注解 ESP和BIOS启动分区不同，5.8.1节中有介绍，它的UUID也不同。

还有一点需要注意，你不能将老的引导装载程序代码放到ESP中，因为这些代码是为BIOS接口写的。你必须提供为UEFI编写的引导装载程序代码。例如，在使用GRUB的时候，你必须安装UEFI版本的GRUB，而非BIOS版本的。另外，你必须向固件注册（声明）新的引导装载程序。

此外，如5.6节中介绍的那样，我们还会面临一些安全启动方面的问题。

5.8.3 GRUB工作原理

让我们来总结一下GRUB，看看它是如何工作的。

(1) BIOS或者固件初始化硬件，在启动存储设备上寻找启动代码。

(2) BIOS和固件运行找到的启动代码，开始GRUB。

(3) 加载GRUB核心。

(4) 初始化GRUB核心，此时GRUB可以读取磁盘和文件系统。

(5) GRUB识别启动分区，在那里加载配置信息。

(6) GRUB为用户提供一个更改配置的机会。

(7) 超时或者用户完成操作以后，GRUB执行配置（执行顺序在5.5.2节有介绍）。

(8) 执行过程当中，GRUB可能会在启动分区中加载额外的代码（模块）。

(9) GRUB执行`boot`命令，以加载和执行配置信息中`linux`命令指定的内核。

由于计算机系统启动机制各异，步骤3和步骤4会非常复杂。最常见的问题是“GRUB核心在哪里？”，对这个问题有三个可能的答案。

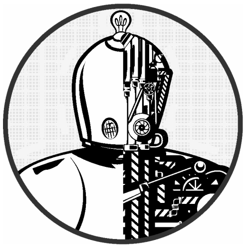
- 部分存储在MBR和第一个分区起始之间的位置。
- 存储在一个常规分区上。
- 存储在一个特殊启动分区上，如GPT启动分区、ESP或者其他地方。

除非你有一个ESP，一般情况下BIOS会从MBR加载512字节，这也是GRUB起始的地方。这一小块信息（从GRUB目录中的boot.img演化而来）还不是核心内容，但是它包含核心信息的起始位置，从此处加载核心。

如果你有ESP，GRUB核心则是一个文件。固件可以让ESP找到并且直接执行GRUB核心，或者是其他操作系统的引导装载程序。

对大多数系统来说，上面的内容只是管中窥豹。在加载和运行内核之前，引导装载程序或许还需要加载一个初始的RAM文件系统映像文件到内容。相关内容请看6.8节中的`initrd`配置参数。不过在此之前让我们先了解下一章的内容：用户空间启动。

第 6 章 用户空间的启动



内核启动第一个用户空间进程是由init开始的。这个点很关键，不仅仅因为此时内存和CPU已经准备就绪，而且你还能看到系统的其余部分是怎样启动运行的。在此之前，内核执行的是受到严格控制的程序序列，由一小撮程序员开发和定义。而用户空间更加模块化，我们容易观察到其中进程的启动和运行过程。对于好奇心强的用户来说，用户空间的启动也更容易修改，不需要底层编程知识即可做到。

用户空间大致按下面的顺序启动：

(1) init；

(2) 基础的底层服务，如udevd和syslogd；

(3) 网络配置；

(4) 中高层服务，如cron和打印服务等；

(5) 登录提示符、GUI及其他应用程序。

6.1 init 介绍

init是Linux上的一个用户空间程序。和其他系统程序一样，你可以在/sbin目录下找到它。它主要负责启动和终止系统中的基础服务进程，但其较新的版本功能更多一些。

Linux系统中，init有以下三种主要的实现版本。

- System V init：传统的顺序init（Sys V，读作“sys-five”），为Red Hat Enterprise Linux和其他的Linux发行版使用。
- systemd：新出现的init。很多Linux发行版已经或者正在计划转向systemd。
- Upstart：Ubuntu上的init。不过在本书编写时，Ubuntu也已经计划转向systemd。

还有一些其他版本的init，特别是在嵌入式系统中。例如，Android就有它自己的init。BSD系统也有它们自己的init，不过在目前的Linux系统中很少见到了（一些Linux发行版通过修改System V init配置来遵循BSD样式）。

init有很多不同版本的实现，因为System V init和其他老版本的程序依赖于一个特定的启动顺序，每次只能执行一个启动任务。这种方式中的依赖关系很简单，然而性能却不怎么好，因为启动任务无法并行。另一个限制是你只能执行启动顺序规定的一系列服务。如果你安装了新的硬件，或者需要启动一个新的服务，该版本的init并不提供一个标准的方法。systemd和Upstart试图解决性能方面的问题，为了加快启动速度，它们允许很多服务并行启动。它们各自的实现差异很大。

- systemd是面向目标的。你定义一个你要实现的目标以及它的依赖条件，systemd负责满足所有依赖条件以及执行目标。systemd还可以将该目标推迟到确实有必要的时候再启动。
- Upstart则完全不同。它能够接收消息，根据接收到的消息来运行任务，并且产生更多消息，然后运行更多任务，以此类推。

systemd和Upstart init系统还为启动和跟踪服务提供了更高级的功能。在传统的init系统中，服务守护进程是通过脚本文件来启动。一个脚本文件负责启动一个守护程序，守护程序脱离脚本自己运行。你需要使用ps命令或其他定制方法来获得守护程序的PID。Upstart和systemd则与此不同，它们可以从一开始就将守护程序纳入管理，提供正在运行程序的更多信息和权限。

因为新的init系统不是基于脚本文件，所以配置起来也相对简单。System V init脚本包含很多相似的命令来启动、停止和重启服务，而在systemd和Upstart中没有这么多冗余，这让你更加专注于服务本身，而非脚本命令。

最后，systemd和Upstart都提供一定程度的即时服务，而不是像System V init那样在启动时开启所有需要的服务。它们根据实际需要开启相应的服务。这并不是什么新概念，传统的inetd守护程序就有，只不过新的实现更为完善。

systemd和Upstart都对System V提供了向后兼容，如支持运行级别（runlevel）的概念。

6.2　System V 运行级别

在Linux系统中，有一组进程自始至终都在运行（如crond和udevd）。System V init中把这个状态叫作系统的*运行级别*，使用数字0~6来表示。系统几乎全程运行在单个运行级别中，但是当你关闭系统的时候，init就会切换到另一个运行级别，有序地终止系统服务，并且通知内核停止。

你可以使用`who -r`命令来查看系统的运行级别。运行Upstart的系统会返回下面的结果：

```
$ who -r
run-level 2 2015-09-06 08:37
```

结果显示系统的当前运行级别是2，还有运行级别起始的时间和日期。

运行级别有几个作用，最主要的是区分系统的启动、关闭、单用户模式和控制台模式等这些不同的状态。例如，Fedora系统一般使用2~4来表示文本控制台，5表示系统将启动图形登录界面。

但是运行级别正在逐渐成为历史。虽然本书涉及的三个init版本都支持它，但systemd和Upstart将其视为已经过时的特性。对它们来说，保留运行级别只是为了启动那些只支持System V init脚本的服务，它们的实现也有很大不同，即便你熟悉其中一个，也未必能够顺势了解另一个。

6.3 识别你的 init

在我们继续之前，你需要确定你系统中的init版本。如果你不确定，可以使用下面的方法查看。

- 如果系统中有目录/usr/lib/systemd和/etc/systemd，说明你有systemd，可以直接跳到6.4节。
- 如果系统中有目录/etc/init，其中包含.conf文件，说明你的系统很可能是Upstart（除非你的系统是Debian 7，那说明你使用的是System V init），可以直接跳到6.5节。
- 如果以上都不是，且你的系统有/etc/inittab文件，说明你可能使用的是System V init，可以跳到6.6节。

如果你的系统安装了帮助手册，你可以查看init(0)帮助手册部分来确认你的init版本。

6.4 systemd

systemd init是Linux上新出现的init实现之一。除了负责常规的启动过程，systemd还包含了一系列的Unix标准服务，如cron和inetd。它借鉴了Apple公司的启动程序。这个init的一个重要特性是：它可以延迟一些服务和操作系统功能的开启，直到需要它们时再开启。

systemd的特性很多，学习起来可能会没有头绪。下面我们列出systemd启动时的运行步骤：

(1) systemd加载配置信息；

(2) systemd判定启动目标，通常是default.target；

(3) systemd判定启动目标的所有依赖关系；

(4) systemd激活依赖的组件并启动目标；

(5) 启动之后，systemd开始响应系统消息（诸如uevent），并且激活其他组件。

systemd并没有一个严格的顺序来启动服务。和现在很多的init系统一样，systemd启动的顺序很灵活，大部分的systemd配置尽量避免需要严格按顺序启动，而是使用其他方法来解决强依赖性问题。

6.4.1 单元和单元类型

systemd最有特色的地方是它不仅仅负责处理进程和服务，还可以挂载文件系统、监控网络套接字和运行时系统等。这些功能我们称之为*单元*，它们的类别称为*单元类型*，开启一个单元称为*激活*。

使用systemd(1)帮助手册可以查看所有的单元类型。这里我们列出了Unix系统启动时需要使用到的单元类型。

- *服务单元*：控制Unix上的传统服务守护进程。

- ❑ 挂载单元：控制文件系统的挂载。
- ❑ 目标单元：控制其余的单元，通常是通过将它们分组的方式。

默认的启动目标通常是一个目标单元，它依赖并组织了一系列的服务和挂载单元。这样你能够很清楚地了解启动过程的情况，还可以使用`systemctl dot`命令来创建一个依赖关系树形图。你会发现这个树状图会很大，因为很多单元默认情况下并不会启动。

图6-1显示了Fedora系统上的default.target单元的部分依赖关系。启动这个单元时，其下的所有单元将被激活。

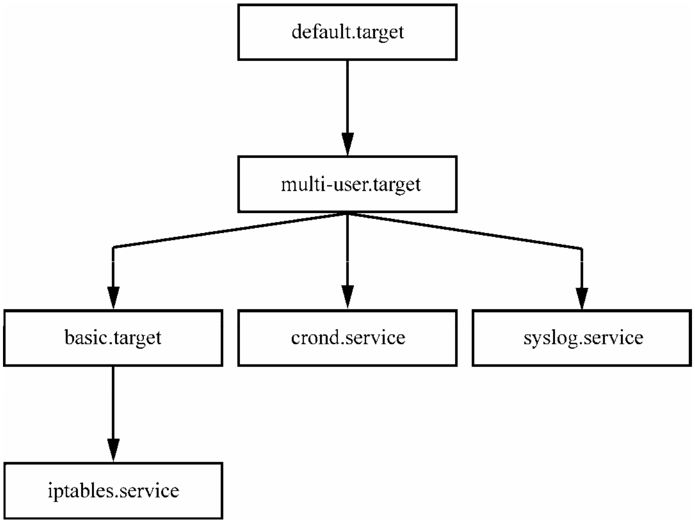


图6-1　单元依赖关系树形图

6.4.2　systemd中的依赖关系

启动时和运行时的依赖关系实际比看上去复杂得多，因为严格的依赖关系非常不灵活。例如，如果你想要在数据库服务启动后显示登录提示符，你可以将登录提示符定义为依赖于数据库服务器。但是如果数据库服务器启动失败，登录提示符也相应地会启动失败，这样你根本没有机会登录系统来修复问题。

Unix的启动任务容错能力很强，一般的错误不会影响那些标准服务的启动。例如，如果一个数据磁盘被从系统中移除，但是/etc/fstab文件仍然存在，文件系统的初始化就会失败，然而这不会太影响系统的正常运行。

为了满足灵活和容错的要求，systemd提供了大量的依赖类型和形式。我们在此按照关键字列出这些类型，但是会在6.4.3节中再详细介绍。基本类型有以下几个。

- ❑ Requires：表示不可缺少的依赖关系。如果一个单元有此类型的依赖关系，systemd会尝试激活被依赖的单元，如果失败，systemd会关闭被依赖的单元。

- Wants：表示只用于激活的依赖关系。单元被激活时，它的Wants类型的依赖关系也会被systemd激活，但是systemd不关心激活成功与否。
- Requisite：表示必须在激活单元前激活依赖关系。systemd会在激活单元前检查其Requisite类型依赖关系的状态。如果依赖关系还没有被激活，单元的启动也会失败。
- Conflicts：反向依赖关系。如果一个单元有Conflict类型的依赖关系，且它们已经被激活，systemd会自动关闭它们。同时启动两个有反向依赖关系的单元会导致失败。

注解 Wants是一种很重要的依赖关系，它不会将启动错误扩散给其他单元。systemd文档鼓励我们尽可能使用这种依赖关系。原因显而易见：它让系统容错性更强，有点像传统的init。

你还可以设定反向的依赖关系。例如，如果要将单元A设定为单元B的Wants依赖，除了在单元B的配置中设置Wants依赖关系，你还可以在单元A的配置中设置反向依赖关系WantedBy。同样的还有RequiredBy。设定反向依赖除了编辑配置文件外，还涉及其他的一些内容，我们将在6.4.3节介绍。

只要你指定一个依赖关系的类型，如Wants或Requires，就可以使用`systemctl`命令来查看单元的依赖关系：

```
# systemctl show -p type unit
```

依赖顺序

到目前为止，依赖关系没有涉及顺序。默认情况下，systemd会在启动单元的同时启动其所有的Requires和Wants依赖组件。理想情况下，我们试图尽可能多、尽可能快地启动服务以缩短启动时间。不过有时候单元必须顺序启动。如图6-1中显示的那样，default.target单元被设定为在multi-user.service之后启动（图中未说明顺序）。

你可以使用下面的依赖关键字来设定顺序。

- Before：当前单元会在Before中列出的单元之前启动。例如，如果Before=bar.target出现在foo.target中，systemd会先启动foo.target，然后是bar.target。
- After：当前单元在After中列出的单元之后启动。

依赖条件

下面我们列出一些systemd中没有使用，但是其他系统使用的依赖条件关键字。

- ConditionPathExists=p: 如果文件路径p存在，则返回`true`。
- ConditionPathIsDirectory=p: 如果p是一个目录，则返回`true`。
- ConditionFileNotEmpty=p: 如果p是一个非空的文件，则返回`true`。

如果单元中的依赖条件为`false`，单元不会被启动，不过依赖条件只对其所在的单元有效。如果你启动的单元中包含依赖条件和其他依赖关系，无论依赖条件为`true`还是`false`，systemd都会启动依赖关系。

其他的依赖关系基本是上述依赖关系的变种，如：RequiresOverridable正常情况下像Requires，

但如果单元手动启动，则像Wants。（可以使用systemd.unit(5)帮助手册查看完整列表。）

至此我们介绍了systemd配置的一些内容，下面我们将介绍单元文件。

6.4.3　systemd配置

systemd配置文件分散在系统的很多目录中，不止一处。但主要是分布在两个地方：系统单元目录（全局配置，一般是/usr/lib/systemd/system）和系统配置目录（局部配置，一般是/etc/systemd/system）。

简单来说，记住这个原则即可：不要更改系统单元目录，因为它由系统来维护。可以在系统配置目录中保存你的自定义设置。在选择更改/usr还是更改/etc时，永远选择/etc。

注解　你可以使用以下命令来查看当前的systemd配置的搜索路径：

```
# systemctl -p UnitPath show
```

该设置信息来自pkg-config。你可以使用以下命令来查看系统单元和配置目录：

```
$ pkg-config systemd --variable=systemdsystemunitdir
$ pkg-config systemd --variable=systemdsystemconfdir
```

单元文件

单元文件是由XDG桌面条目规范（XDG Desktop Entry Specification，.desktop文件，类似Windows中的.ini文件）演变而来，[]中的是区块名称，每个区块包含变量和变量值。

我们来看一看Fedora系统中/usr/lib/systemd/system目录下的media.mount单元文件。该文件是针对/media tmpfs文件系统的，这个目录负责可移动媒体的挂载。

```
[Unit]
Description=Media Directory
Before=local-fs.target

[Mount]
What=tmpfs
Where=/media
Type=tmpfs
Options=mode=755,nosuid,nodev,noexec
```

上面有两个区块，区块`[Unit]`包含单元信息和依赖信息，该单元被设定为在local-fs.target单元之前启动。

区块`[Mount]`表示该单元一个挂载单元，包含挂载点信息、文件系统类型和挂载选项（参考4.2.6节）。`What=`变量定义了挂载的设备或者设备的UUID。本例中是tmpfs，因为它没有对应的设备。（可以参考systemd.mount(5)帮助手册查看全部挂载单元选项。）

其他单元配置文件也很简单。例如，下面的服务单元文件sshd.service启动安全登录shell：

```
[Unit]
Description=OpenSSH server daemon
After=syslog.target network.target auditd.service

[Service]
EnvironmentFile=/etc/sysconfig/sshd
ExecStartPre=/usr/sbin/sshd-keygen
ExecStart=/usr/sbin/sshd -D $OPTIONS
ExecReload=/bin/kill -HUP $MAINPID

[Install]
WantedBy=multi-user.target
```

这是一个服务目标，详细信息在[Service]区块中，包括服务如何准备就绪、如何启动和重新启动。你可以在systemd.exec(5)帮助手册中查看完整的列表，另外在6.4.6一节中也有介绍。

6

开启单元和[Install]区块

sshd.service单元文件中的[Install]区块很重要，因为它告诉我们怎样使用systemd的WantedBy和RequiredBy依赖关系。它能够开启单元，同时不需要任何对配置文件的更改。正常情况下systemd会忽略[Install]部分。然而在某种情况下，系统中的sshd.service被关闭，你需要开启它。你开启一个单元的时候，systemd读取[Install]区块。这时，开启sshd.service单元就需要systemd去查看multi-user.target的WantedBy依赖关系。相应地，systemd在系统配置目录中创建一个符号链接来指向sshd.service，如下所示：

```
ln -s '/usr/lib/systemd/system/sshd.service' '/etc/systemd/system/multi-user.
target.wants/sshd.service'
```

注意，该符号链接创建于被依赖的单元所对应的子目录中（multi-user.target的子目录）。

[Install]区块通常对应系统配置目录（/etc/sytemd/system）中的.wants和.requires目录。不过在单元配置目录（/usr/lib/systemd/system）中也有.wants目录，你可以在单元文件中创建无关[Install]区块的符号链接。这种方法让你可以不用更改单元文件就能够加入依赖关系，因为单元文件有可能被系统更新覆盖。

注解 开启（enable）单元和激活（activate）单元不同。开启单元是指你将其安装到systemd的配置中，做一些在重启后会保留的非永久性的更改。不过你并非总是需要明确地开启单元。如果单元文件包含[Install]区块，你就需要通过systemctl enable来开启。否则单元文件本身就足以完成开启。当你使用systemctl start来激活单元时，你只是在当前运行时环境中打开它。此外，开启单元并不意味着激活单元。

变量和说明符

sshd.service单元文件还包含了一些变量，如systemd传递过来的$OPTIONS和$MAINPID环境变量。当你使用systemctl激活单元时，可以用$OPTIONS变量为sshd设定选项。$MAINPID是被追踪的

服务进程（参考6.4.6节）。

说明符（specifier）是单元文件中另一种类似变量的机制，前缀为%。例如，%n代表当前单元的名称，%H代表当前主机名。

> **注解**　单元名中可以包含一些说明符。你可以为单元文件使用参数来启动一个服务的多个实例，例如在tty1和tty2上运行的getty进程。你可以在单元文件名末尾加上@来使用说明符。比如对getty来说，你可以创建一个名为getty@.service的单元文件，该文件代表getty@tty1和getty@tty2这样的单元。@之后的内容我们称为实例，在单元文件执行时，systemd展开%I说明符。在大多数运行systemd的系统中，你可以找到getty@.service并看看它实际是怎样工作的。

6.4.4　systemd操作

我们主要通过systemctl命令与systemd进行诸如激活服务、关闭服务、显示状态、重新加载配置等的交互操作。

最基本的命令主要用于获取单元信息。例如，使用list-units命令来显示系统中所有激活的单元（实际上这是systemctl的默认命令，你不需要指定list-units部分）：

```
$ systemctl list-units
```

输出结果是典型的Unix列表形式，如下所示：

```
UNIT              LOAD    ACTIVE  SUB        JOB DESCRIPTION
media.mount       loaded  active  mounted    Media Directory
```

该命令的输出有很多信息，因为系统中有大量的激活单元。由于systemctl会将长单元名截短，可以使用--full选项来查看完整的单元名。使用--all选项查看所有单元（包括未激活的）。

另一个很有用的systemctl操作是获得单元的状态信息。下例是典型的状态命令和输出结果。

```
$ systemctl status media.mount
media.mount - Media Directory
          Loaded: loaded (/usr/lib/systemd/system/media.mount; static)
          Active: active (mounted) since Wed, 13 May 2015 11:14:55 -0800;
37min ago
           Where: /media
            What: tmpfs
         Process: 331 ExecMount=/bin/mount tmpfs /media -t tmpfs -o
mode=755,nosuid,nodev,noexec (code=exited, status=0/SUCCESS)
          CGroup: name=systemd:/system/media.mount
```

这里输出的信息比传统的init系统多很多：不仅仅是该单元的状态，还有执行挂载的命令、

PID和退出状态。

输出结果中最有意思的信息是控制组名。在上例中，控制组除了systemd:/system/media.mount这个名称之外并不包括其他信息，因为单元处理过程这时已经终止了。然而如果你从NetworkManager.service这样的服务单元获得状态信息，你就能看到控制组的进程树结构。你可以使用`system-cgls`命令来查看控制组。详细内容我们将在6.4.6节介绍。

status命令还显示最新的单元日志信息。你可以使用以下命令查看完整的单元日志：

```
$ journalctl _SYSTEMD_UNIT=unit
```

（它的语法有一点奇怪，因为`journalctl`不仅用来显示systemd单元日志，还用来显示其他日志。）

你可以使用`systemd start`、`stop`和`restart`命令来激活、关闭和重启单元。但如果你更改了单元配置文件，你可以使用以下两种方法让systemd重新加载文件。

- ❑ `systemctl reload unit`：只重新加载unit单元的配置。
- ❑ `systemctl daemon-reload`：重新加载所有的单元配置。

在systemd中，我们将激活、关闭和重启单元称为任务（job），它们本质上是对单元状态的变更。你可以用以下命令来查看系统中的当前任务：

```
$ systemctl list-jobs
```

如果已经运行了一段时间，系统中可能已经没有任何激活的任务，因为所有激活工作应该已经完成。然而，在系统启动时，如果你很快登录系统，你可以看到一些单元正在慢慢被激活，如下所示：

```
JOB UNIT                      TYPE          STATE
  1 graphical.target          start         waiting
  2 multi-user.target         start         waiting
 71 systemd-...nlevel.service start         waiting
 75 sm-client.service         start         waiting
 76 sendmail.service          start         running
120 systemd-...ead-done.timer start         waiting
```

上例中的任务76是sendmail.service单元，它的启动花了很长时间。其他的任务处于等待状态，它们很有可能是在等待任务76。任务76在sendmail.service启动完成时会终止，其余的任务会继续，直到任务列表完全清空。

注解 任务这个词可能不太好理解，特别是我们在本章介绍过的Upstart也使用它来代表systemd中的单元。有一点需要注意，单元能够使用任务来启动，任务完成后会终止。单元，特别是服务单元，在没有任务的情况下也能够被激活和运行。

我们将在6.7节介绍如何关闭和重启系统。

6.4.5　在systemd中添加单元

在systemd中添加单元涉及创建和激活单元文件，有时候还需要开启单元文件。将单元文件放入系统配置目录/etc/systemd/system，这样你就不会将它们与系统自带的配置混淆起来，它们也不会被系统更新覆盖了。

创建一个什么都不做的目标单元很简单，你可以自己试一试。我们来创建两个目标，其中一个依赖于另一个。

(1) 创建一个名为test1.target的单元：

```
[Unit]
Description=test 1
```

(2) 创建test2.target，其依赖于test1.target：

```
[Unit]
Description=test 2
Wants=test1.target
```

(3) 激活test2.target单元（test1.target作为依赖关系也会被激活）：

```
# systemctl start test2.target
```

(4) 验证两个单元都被激活：

```
# systemctl status test1.target test2.target
test1.target - test 1
          Loaded: loaded (/etc/systemd/system/test1.target; static)
          Active: active since Thu, 12 Nov 2015 15:42:34 -0800; 10s ago

test2.target - test 2
          Loaded: loaded (/etc/systemd/system/test2.target; static)
          Active: active since Thu, 12 Nov 2015 15:42:34 -0800; 10s ago
```

注解　如果单元文件中包含[Install]区块，你需要在激活前开启它：

```
# systemctl enable unit
```

你可以在上例中运行上面这个命令。将依赖关系从test2.target中去掉，在test1.target加上[Install]区块WantedBy=test2.target。

删除单元

使用以下步骤来删除单元。

(1) 必要时关闭单元：

```
# systemctl stop unit
```

(2) 如果单元中包含[Install]区块，则通过关闭单元来删除依赖的符号链接：

```
# systemctl disable unit
```

(3) 这时你就可以删除单元文件了。

6.4.6　systemd进程跟踪和同步

对于启动的进程，systemd需要掌握大量的信息和控制权。最大的问题是启动一个服务的方式有多种。这样导致的后果是，可能会叉分（fork）出许多新实例，甚至还可能会将其作为守护进程并且同原始进程脱离开。

为了减小开发人员或系统管理员创建单元文件所需的工作量，systemd引进了控制组，它是Linux内核的一个可选特性，为的是提供更好的进程跟踪。如果你接触过Upstart就知道，为了找到一个服务的主进程，需要做一些额外的工作。在systemd中，你不需要担心一个进程被叉分了多少次，只需要知道它能不能被叉分。你可以在服务单元文件中使用Type选项来定义其启动行为。启动行为有以下两种。

- Type=simple：服务进程不能叉分。
- Type=forking：systemd希望原始的服务进程在叉分后终止，原始进程终止时，systemd视其为服务准备就绪。

Type=simple选项并不负责服务花多长时间启动，systemd也不知道何时启动该服务的依赖关系。解决这个问题的一个办法是使用延时启动（参考6.4.7节）。不过我们可以使用Type来让服务就绪时通知systemd。

- Type=notify：服务在就绪时向systemd发送通知（使用sd_notifiy()函数）。
- Type=dbus：服务在就绪时向D-Bus（Desktop Bus）注册自己。

另外还有一个服务启动类型是Type=oneshot，其中服务进程在完成任务后会彻底终止。对于这种启动类型，基本上你都需要加上RemainAfterExit=yes选项来确保systemd在服务进程终止后仍然将服务状态视作激活。

最后还有一个类型Type=idle，意思是在当前没有任何激活任务的情况下，systemd才会激活该服务。这个类型主要是用于等其他服务都启动完成后，再启动制定的服务，这样可以减轻系统负载，还可以避免服务启动过程之间的交叉。（请记住，服务启动后，启动服务的systemd任务即终止。）

6.4.7　systemd的按需和资源并行启动

systemd的一个最主要的特性是它可以延迟启动单元，直到它们真正被需要为止。配置方式如下所示。

(1) 为系统服务创建一个systemd单元（单元A）。

(2) 标识出单元A需要为其服务提供的系统资源，如网络端口、网络套接字或者设备。

(3) 创建另一个systemd单元（单元R）来表示该资源。它有特殊的单元类型，如套接字单元、

路径单元和设备单元。

其运行步骤如下所示。

(1) 单元R激活的时候，systemd对其资源进行监控。

(2) systemd将阻止所有对该资源的访问，对该资源的输入会被缓冲。

(3) systemd激活单元A。

(4) 当单元A启动的服务就绪时，其获得对资源的控制，读取缓冲的输入，然后正常运行。

同时，有以下几个问题需要考虑。

- 必须确保资源单元涵盖了服务提供的所有资源。通常这不是大问题，因为大部分服务只有一个单一的访问点。
- 必须确保资源单元与其代表的服务单元之间的关联。这可以是显式或者是隐式，有些情况下，systemd可以有许多选项使用不同的方式来调用服务单元。
- 并非所有的服务器都能够和systemd提供的单元进行交互。

如果你了解诸如`inetd`、`xinetd`和`automount`这样的工具，你就知道它们之间有很多相似的地方。事实上这个概念本身没什么新奇之处（实际上systemd包含了对automount单元的支持）。我们将在“套接字单元和服务”一节中介绍一个套接字的例子。但是首先来让我们看看系统启动过程中资源单元的作用。

使用辅助单元优化启动

systemd在激活单元时通常会试图简化依赖关系和缩短启动时间。这类似于按需启动，其中辅助单元代表服务单元所需的资源，不同的地方是systemd在激活辅助单元之后立即启动服务单元。

使用该模式的一个原因是，一些关键的服务单元如syslog和dbus需要一些时间来启动，有许多单元依赖于它们。然而，systemd能快速地提供单元所需的重要资源（如套接字单元），因此它不仅能够快速启动这个关键单元，还能够启动依赖于它的其他单元。关键单元就绪后，就能获得其所需资源的控制权。

图6-2显示了这一切在传统系统中是如何工作的。在启动时间线上，服务E提供了一个关键资源R。服务A、B和C依赖于这个资源，必须等待服务E先启动。系统启动时，要启动服务C需要很长一段时间。

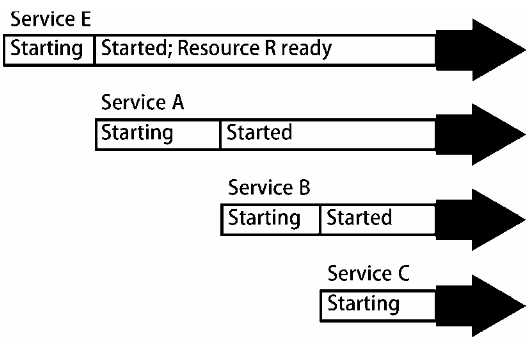


图6-2　启动时间顺序和资源依赖关系

图6-3显示与图6-2对应的systemd的启动配置。有服务单元A、B、C、E和一个新的单元R代表单元E提供的资源。因为systemd在单元E启动时能够为单元R提供一个接口，单元A、B、C和E能够同时启动。单元E在单元R就绪时接管。（有意思的是，如单元B配置所示，单元A、B和C并不需要在它们结束启动前显式访问单元R。）

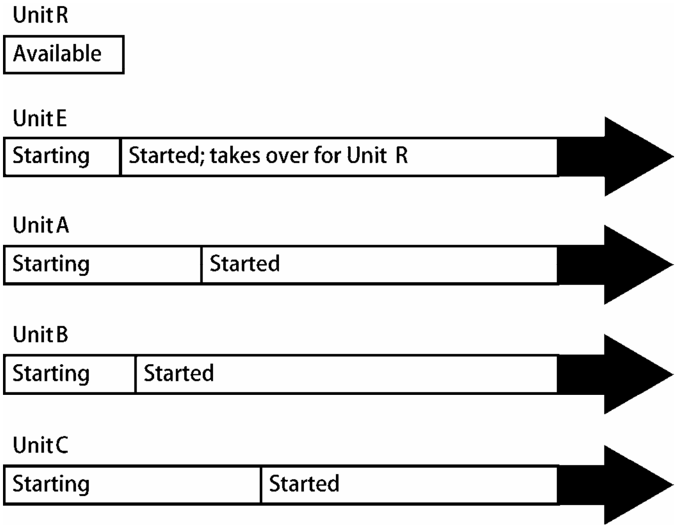


图6-3　systemd启动时间顺序和资源单元

注解　当并行启动时，系统有可能会因为大量单元同时启动而暂时变慢。

本例中虽然并没有创建按需启动的单元，但是仍然用到了按需启动的特性。在日常操作中，你可以在运行systemd的系统中查看syslog和D-Bus配置单元。它们大都是以这样的方式来并行启动的。

套接字单元和服务实例

下面我们看一个实例，这是一个简单的网络服务，使用一个套接字单元。本节内容涉及TCP、网络端口和网络监听，这些内容我们将在第9章和第10章中介绍，如果你现在觉得不好理解，可以暂时跳过。

本例中服务的功能是，当网络客户端连接服务时，服务将客户端发送的数据原样发送回客户端。服务单元使用TCP端口22222来监听请求。我们将此服务命名为回音服务，它通过一个套接字单元启动，单元内容如下：

```
[Unit]
Description=echo socket

[Socket]
```

```
ListenStream=22222
Accept=yes
```

注意，在单元文件中并没有提及该套接字支持的服务单元，那么与之相关的服务单元文件在哪里呢？

服务单元文件名是echo@.service，两者是通过命名规范来建立关联的。如果服务单元文件名和套接字单元文件名（echo.socket）的前缀一样，systemd会在套接字单元有请求时激活服务单元。本例中，当echo.socket有请求时，sytemd会创建一个echo@.service的实例。

以下是echo@.service单元文件：

```
[Unit]
Description=echo service

[Service]
ExecStart=-/bin/cat
StandardInput=socket
```

注解　如果你不喜欢使用前缀来隐式地激活单元，或者你需要激活有不同前缀的单元，你可以在单元中定义使用的资源来显式地激活。例如，在foo.service中加入Socket=bar.socket，让bar.socket为foo.service提供它的套接字资源。

你可以使用下面的命令来启动服务：

```
# systemctl start echo.socket
```

你可以使用telnet命令连接本地端口22222来测试该服务是否运行，键入任意内容然后回车，服务会将你的输入内容原样输出：

```
$ telnet localhost 22222
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi there.
Hi there.
```

按CTRL-]，然后CTRL-D来结束服务，使用以下命令停止套接字单元：

```
# systemctl stop echo.socket
```

实例和移交

echo@.service单元支持多个实例同时运行，其文件名中含有@（在前文注解中我们介绍过，@代表参数化）。那么我们为什么需要多个实例呢？因为很可能会有多个网络客户端同时连接到

服务，每个连接需要一个专属的实例。

因为echo.socket中的Accept选项，服务单元必须支持多个实例。该选项告诉systemd在监听端口的同时接受呼入的连接请求，并将连接传递给服务单元，每个连接是一个单独的实例。每个实例将连接作为标准输入，从中读取数据，不过实例并不需要知道数据是来自网络连接。

注解 大多数网络连接除了需要与标准输入输出的简单接口外，还需要更多的灵活性。所以本例中的echo@.service只是一个很简单的例子，实际的网络服务要复杂得多。

虽然服务单元可以完成接受连接的所有工作，但此时它的文件名中并不包含@。在这种情况下，它会获得对套接字的全部控制权。systemd在服务单元完成任务前不会试图去监听该网络端口。

由于各种资源和选项的差异，我们无法为资源移交给服务单元这个过程总结出一个简单的模式。并且这些选项的文档也分散在帮助手册中的各个地方。关于资源相关单元的文档，你可以查阅systemd.socket(5)、systemd.path(5)和systemd.device(5)。关于服务单元的一个经常被忽略的文档是systemd.exec(5)，它描述了服务单元在激活时如何获得资源的情况。

6.4.8 systemd的System V兼容性

systemd中有一个特性让其有别于其他的新一代init系统，就是对于System V兼容init脚本启动的服务，systemd会尽量完全地进行监控。它的工作流程如下所示。

(1) systemd首先激活runlevel<N>.target，其中N是运行级别。

(2) systemd为/etc/rc<N>.d中的每一个符号链接在/etc/init.d中标识出对应脚本。

(3) systemd将脚本名和服务单元关联起来（例如：/etc/init.d/foo对应foo.service）。

(4) systemd根据rc<N>.d中的名称，激活服务单元，使用参数start或者stop运行脚本。

(5) systemd尝试关联脚本进程和服务单元。

由于systemd根据服务单元来建立关联，你可以使用systemctl来重启服务和查看其状态。不过System V兼容模式仍然按顺序执行init脚本。

6.4.9 systemd辅助程序

使用systemd的时候，你可能会注意到/lib/systemd目录中有大量的程序，它们主要是单元的支持程序。例如，作为systemd的一个组成部分，udevd对应的程序文件是systemd-udevd。此外，程序文件systemd-fsck是作为systemd和fsck的“中间人”而存在。

这些程序很多都有标准系统工具程序所不具备的消息通知机制。它们通常运行标准系统工具程序，然后将执行结果通知给systemd。（毕竟重新实现systemd中的fsck是不太现实的。）

注解　这些程序都是使用C编写的，因为systemd的目的之一就是减少系统中脚本文件的数量。这究竟是不是个好主意还有很多争论（毕竟它们中很多都可以使用脚本来实现），不过最重要的是它们能够稳定、安全、快速地运行，至于用脚本还是C来编写则是次要的。

如果你在/lib/systemd中看到一个不认识的程序，你可以查阅帮助手册。帮助手册很有可能不仅提供该程序的信息，还提供它的单元类型。

如果你的系统中没有Upstart，或者你不感兴趣，你可以跳到6.6节去了解System V init进程。

6.5　Upstart

init的Upstart版本主要涉及任务和事件。任务是启动和运行时Upstart执行的操作（如系统服务和配置），事件是Upstart从自身或者其他进程（如udevd）接收到的消息。Upstart通过启动任务的方式来响应消息。

为了理解它的工作原理，我们来看看启动udevd守护进程的udev任务。它的配置文件通常是/etc/init/udev.conf，其中包含下面的内容：

```
start on virtual-filesystems
stop on runlevel [06]
```

它们表示Upstart在接收到virtual-filesystems事件时启动udev任务，在接收到带有参数0或者6的运行级别事件后停止。

事件和它们的参数有很多变种。例如，Upstart能响应任务状态触发的消息，如udev任务触发的started udev事件。在详细介绍任务之前，我们先介绍一下Upstart大致的工作原理。

6.5.1　Upstart初始化过程

Upstart的启动步骤如下：

(1) 加载自身配置和/etc/init中的任务配置文件；

(2) 产生startup事件；

(3) 启动那些响应startup事件的任务；

(4) 这些任务产生各自的事件，触发更多的任务和事件。

在完成所有正常启动相关的任务之后，Upstart继续监控和响应系统运行时产生的事件。

大多数Upstart的安装步骤如下所示。

(1) 在Upstart响应startup事件所运行的任务中，mountall是最重要的一个。它为系统挂载所有必要的本地和虚拟文件系统，以保障系统其他部分能够运行。

(2) mountall任务会产生一些事件，包括filesystem、virtual-filesystems、local-filesystems、remote-filesystems和all-swaps等。它们表示这些重要的文件系统已经挂载完毕并准备就绪。

(3) 为了响应这些事件，Upstart启动一系列的服务。例如，为virtual-filesystems事件启动udev，为local-filesystems事件启动dbus。

(4) 在local-filesystems事件和udevd就绪时，Upstart启动network-interfaces任务。

(5) network-interfaces任务产生static-network-up事件。

(6) Upstart为响应filesystem和static-network-up事件运行rs-sysinit任务。该任务负责维护系统当前的运行级别，在第一次没有运行级别启动时，它通过产生runlevel事件将系统切换到默认的运行级别。

(7) 为了响应runlevel事件和新的运行级别，Upstart运行系统中的其他大部分启动任务。

这个过程可能会变得很复杂，因为事件产生的源头并不总是很清晰。Upstart本身只产生几个事件，其余的都来自任务。任务配置文件通常都声明了它们会产生的事件，但是产生事件的细节往往不在Upstart任务配置文件中。

为了弄清事情的本质，通常你需要深入挖掘。以static-network-up事件为例，network-interface.conf任务配置文件声明了它会产生该事件，但是没说从哪里产生。我们发现事件来自于ifup命令，它是由该任务使用脚本/etc/network/if-up.d/upstart来初始化网络接口时运行的。

注解 虽然所有的这些过程都有文档（ifup.d目录在帮助手册interfaces(5)中能找到，ifup(8)帮助手册引用了这部分内容），但是光靠阅读文档来理解整个工作原理并不是一件简单的事。更快的方法是使用grep在配置文件中搜索事件名称来查看相关的内容。

Upstart的一个问题是没有办法清晰地查看事件的来龙去脉。你可以将它的日志优先级设置为debug，这样你可以看到所有的日志信息（通常在/var/log/syslog中），但是大量的信息会让人难以查找事件的相关内容。

6.5.2 Upstart任务

Upstart的/etc/init配置目录中的每个文件都对应一个任务，每个任务的主配置文件都有.conf后缀。例如，/etc/init/mountall.conf即针对mountall任务。

Upstart任务主要分为以下两大类。

- Task任务：这些任务会在某一明确的时刻终止。例如，mountall就是一个task任务，它在挂载完文件系统后终止。
- Service任务：这些任务没有明确的终止时间。像udevd这样的守护服务进程、数据库服务器和网络服务器都属于service任务。

还有第三种任务叫抽象任务，也可以把它们看成是虚拟的service任务。它们只存在于Upstart中，本身什么都不运行，不过有时候其他任务的管理工具会使用它们产生的事件来启动和停止任务。

查看任务

你可以使用initctl命令来查看Upstart任务和状态。下面的命令用来查看整个系统的运行状态：

```
$ initctl list
```

它输出的内容很多，我们来看两个比较有代表性的任务：

```
mountall stop/waiting
```

上例显示，mountall的task任务状态为stop/waiting，意思是并未运行。（但很遗憾，到本书成书为止，你还不能根据状态信息来确定任务是否已经运行过，因为stop/waiting状态还有可能表示任务从来未被运行过。）

有关联进程的service任务的状态显示如下：

```
tty1 start/running, process 1634
```

上例表示tty1任务正在运行，与之关联的进程ID为1634。（不是所有的service任务都有关联的进程。）

注解　如果你知道任务的名称，你可以直接使用initctl status job查看任务状态。

initctl输出结果中的状态（如stop/waiting）可能会让人有些不解。左边（/之前）的部分是目标，或者说是任务将要达到的状态，如start或stop。右边的部分是任务的当前状态，如waiting或running。例如上面的例子中，tty1任务的状态是start/running，意思是它的目标是start，而状态running表示它已经启动成功。（对于service任务来说，状态running只是象征性的。）

mountall则有一些不同，因为task任务不持续运行。状态stop/waiting通常表示任务已经启动并且执行完毕。在执行完毕时，它从目标start切换至stop，等待来自Upstart的后续指令。

但之前提过，状态为stop/waiting的任务也可能从未启动过。所以除非你开始调试功能来查看日志，否则单从状态上无法分辨任务是已经执行完毕还是从未启动。详见6.5.5节。

注解　你无法查看那些通过Upstart的System V兼容特性启动的任务。

任务状态转换

任务状态有很多种，但是它们之间的转换方式很固定。例如，通常任务是按照下列步骤启动的。

(1) 所有的任务起始状态为stop/waiting。

(2) 当用户或者系统事件触发任务时，任务目标从stop变为start。

(3) Upstart将任务状态从waiting变为starting，从而当前状态为start/starting。

(4) Upstart产生starting job事件。

(5) 任务执行starting状态的相关操作。

(6) Upstart将任务状态从starting变为pre-start，并产生pre-start job事件。

(7) 任务经过数次状态转换，最终变为running状态。

(8) Upstart产生started job事件。

任务的终止也涉及一系列类似的状态转换和事件。（可以查阅upstart-events(7)帮助手册。）

6.5.3 Upstart配置

我们来看一下这两个配置文件：一个是task任务mountall，另一个是service任务tty1。和所有的Upstart配置文件一样，它们存放在目录/etc/init下，文件名为mountall.conf和tty1.conf。配置文件由更小的节（stanza）组成。每个节开头是一个关键字，诸如description和start等。

首先我们可以打开mountall.conf文件，在第一个节中寻找以下内容：

```
description     "Mount filesystems on boot"
```

该行包含了对任务的简短文字描述。

接下来的几个节描述mountall任务如何启动：

```
start on startup
stop on starting rcS
```

第一行告诉Upstart在接收到startup事件（Upstart产生的第一个事件）时启动任务。第二行告诉Upstart在接收到rcS事件（此时系统进入单用户模式）时终止任务。

下面两行内容告诉Upstart任务mountall的运行方式：

```
expect daemon
task
```

task告诉Upstart它是一个task任务，因此任务会在某一时刻完成。expect则有一些复杂，它表示mountall任务会复制一个守护进程，且独立于原来的任务脚本运行。Upstart需要知道这些信息，因为它需要知道守护进程何时结束，以便发送消息通知mountall任务已经结束。（相关内容我们将稍后介绍。）

mountall.conf文件中还有一些emits节，用来说明任务会产生哪些事件：

```
emits virtual-filesystems
emits local-filesystems
emits remote-filesystems
emits all-swaps
emits filesystem
emits mounting
emits mounted
```

> **注解**　我们在6.5.1节中提到过，这些所谓的“节”并不是真正的事件源，你需要在任务脚本中去寻找它们。

你还可能会看到console节，它表示Upstart需要将任务信息输出到哪里：

```
console output
```

output参数表示Upstart将mountall的任务信息输出到系统控制台。

接下来你会看到任务的细节，它是一个script节。

```
script
    . /etc/default/rcS
    [ -f /forcefsck ] && force_fsck="--force-fsck"
    [ "$FSCKFIX" = "yes" ] && fsck_fix="-fsck-fix"

    # set $LANG so that messages appearing in plymouth are translated
    if [ -r /etc/default/locale ]; then
        . /etc/default/locale
        export LANG LANGUAGE LC_MESSAGES LC_ALL
    fi

    exec mountall --daemon $force_fsck $fsck_fix
end script
```

这是一个shell脚本（参见第11章），主要做一些预备工作，如设置本地化参数、判断是否需要fsck。其底部的exec mountall命令执行真正的操作。这个命令的功能是挂载文件系统，并且在结束时产生任务需要的事件。

Service任务tty1

Service任务tty1就简单得多，它控制一个虚拟控制台登录提示符。它的配置文件tty1.conf如下：

```
start on stopped rc RUNLEVEL=[2345] and (
            not-container or
            container CONTAINER=lxc or
            container CONTAINER=lxc-libvirt)

stop on runlevel [!2345]

respawn
exec /sbin/getty -8 38400 tty1
```

该任务最复杂的部分在于它的启动，不过现在让我们先来看下面这一行：

```
start on stopped rc RUNLEVEL=[2345]
```

它告诉Upstart在接收到stopped rc事件时（由Upstart在rc task任务执行完毕时产生）激活任

务。为了满足该条件，rc任务还必须将RUNLEVEL环境变量设置为2~5之间的某个值（参考6.5.6节）。

注解 其他基于运行级别的任务没有这么多条件，例如：

```
start on runlevel [2345]
```

本例和前例的区别是启动时机不同。本例中任务在runlevel被设置时启动，而前例则需要等到System V相关任务结束才启动。

容器（container）配置文件之所以存在，是因为Upstart不仅仅在硬件系统的内核上运行，还能够在虚拟环境和容器中运行。一些环境中没有虚拟控制台，无法运行getty。

停止tty1任务很简单：

```
stop on runlevel [!2345]
```

该文本行告诉Upstart当运行级别不是2～5之间的值的时候停止任务（例如在系统关闭时）。

最底部的exec文本行是这样一个命令：

```
exec /sbin/getty -8 38400 tty1
```

它类似于你在mountall任务中见到的script节，区别是tty1任务的设置很简单，一行命令足够。该命令在/dev/tty1上运行getty登录提示符程序，它是系统的第一个虚拟控制台（可以在图形界面中按CTRL-ALT-F1打开）。

respawn文本行告诉Upstart任务终止时重新启动tty1任务。当你从虚拟控制台退出时，Upstart启动一个新的getty登录提示符。

以上是基础的Upstart配置。你可以在帮助手册init(5)和在线文档找到更详细的内容。有一个需要特别提及的expect节，将在稍后介绍。

进程跟踪和Upstart的expect节

Upstart能在任务启动后跟踪它们的进程（因此它才能执行终止和重启），它的任务是知道与每个任务相关联的进程。这可能会有些困难，因为在传统的Unix启动方式中，进程从其他进程产生分支成为守护进程，任务对应的主进程也许在产生一两个分支后才启动。如果没有一个好的跟踪机制，Upstart很难完成任务的启动，也很容易跟踪到错误的PID。

我们使用expect来告诉Upstart有关任务执行的细节，有以下4种情况。

- 没有expect：表示任务的主进程不产生分支，可直接跟踪主进程。
- expect fork：表示进程产生一次分支，跟踪分支进程。
- expect daemon：表示进程产生两次分支，跟踪第二个分支。
- expect stop：任务的主进程会发出SIGSTOP信号，表示已经准备就绪。（这种情况很少见。）

对于Upstart和systemd这些新版本的init而言，最好的是第一种情况（没有 expect），因为任务

的主进程不需要包含关于自身启动和关闭的机制。另一方面，它不需要考虑从当前终端产生分支和分离，这些麻烦的东西是Unix开发者很长时间以来都需要处理的。

很多传统的服务守护进程都包含调试选项，让主进程不要产生分支。Secure Shell守护进程sshd及其选项-D是其中一个例子。在下例中，/etc/init/ssh.conf的启动代码中包含启动sshd的一个简单配置，它防止了进程过快的再生，并且清除了很多误导人的stderr输出：

```
respawn
respawn limit 10 5
umask 022

# 'sshd -D' leaks stderr and confuses things in conjunction with 'console log'
console none
--snip--

exec /usr/sbin/sshd -D
```

对于包含expect的任务来说，expect fork很常见。例如下面是/etc/init/cron.conf的启动部分：

```
expect fork
respawn

exec cron
```

这样简洁的启动配置通常显示的是稳定安全的守护进程。

注解　关于expect的内容，推荐到upstart.ubuntu.com站点阅读更多的文档，因为它和进程生命周期直接相关。比如，你可以使用strace命令来跟踪一个进程和它的系统调用，包括fork()。

6.5.4　Upstart操作

除了6.5.2节中介绍的list和status命令，你还可以用initctl工具来操控Upstart及其任务。详细信息可阅读帮助手册initctl(8)，但现在先来看一些基础内容。

使用initctl start来启动Upstart任务：

```
# initctl start job
```

使用initctl stop来停止任务：

```
# initctl stop job
```

重启任务使用以下命令：

```
# initctl restart job
```

如果想向Upstart发出事件，你可以运行：

```
# initctl emit event
```

你还可以通过在event后加上key=value参数来向事件传递环境变量。

> **注解** 你无法单独启动或者停止由Upstart的System V兼容模式启动的服务。可以参见6.6.1节来了解在System V init脚本中怎么做。

关闭Upstart任务以禁止其启动时运行的方法有很多种，但可维护性最高的一种是确定任务配置文件的文件名（通常是/etc/init/<job>.conf），然后创建一个/etc/init/<job>.override文件，仅包含下面一行内容：

```
manual
```

这样，唯一能够启动任务的方式就成了运行initctl start job。

这个方法的好处是很容易撤销，如果要在启动时重新开启任务，只需要删除.override文件即可。

6.5.5 Upstart日志

Upstart中有两种基本的日志类型：service任务日志和由Upstart自己产生的系统诊断信息。Service任务日志记录脚本和运行服务的守护进程所产生的标准输出和标准错误输出内容。这些信息保存在/var/log/upstart中，作为服务产生的系统日志的一种补充（关于系统日志我们将在第7章详细介绍）。这些日志中的内容很难分类，比较常见的内容是启动和关闭消息，以及一些紧急错误消息。很多服务根本不产生信息，因为它们将所有日志记录到系统日志或者它们自己的日志中。

Upstart自带的系统诊断日志包含其何时启动和重新加载，还有任务和事件的相关信息。该日志使用内核的系统日志工具。在Ubuntu上，它们通常保存在/var/log/kern.log和/var/log/syslog中。

默认情况下，Upstart仅仅记录很少的日志，甚至不记录日志。如果要查看日志，你必须更改Upstart日志的优先级。默认优先级是message。可以将优先级设置为info来记录事件和任务信息：

```
# initctl log-priority info
```

需要注意的是，该设置会在系统重启后重置。你可以在启动参数中加上--verbose参数，让Upstart在系统启动时记录所有信息，具体内容可参考5.5节。

6.5.6 Upstart运行级别和System V兼容性

到目前为止，我们介绍了Upstart如何支持System V运行级别，也说过它能够将System V启动脚本作为任务来启动。下面是它在Ubuntu上运行的详细情况。

(1) rc-sysinit任务运行，通常是在接收到`filesystem`和`static-network-up`事件后。在其运行之前，运行级别没有设置。

(2) rc-sysinit任务决定进入哪一个运行级别。通常运行级别是默认，也有可能从较老的/etc/inittab文件或者内核参数（/proc/cmdline）中获得。

(3) rc-sysinit任务运行`telinit`来切换运行级别。该命令产生一个`runlevel`事件，在`RUNLEVEL`环境变量中设置运行级别值。

(4) Upstart接收到`runlevel`事件。每个运行级别都配置有一系列的任务来响应`runlevel`事件，由Upstart负责启动。

(5) rc是由运行级别激活的任务之一，它负责运行System V启动。和System V init一样，rc运行/etc/init.d/rc（见6.6节）。

(6) rc任务停止后，Upstart在接收到`stopped rc`事件后启动一系列其他任务（如"Service任务tty1"一节中介绍过的tty1）。

请注意，虽然Upstart将运行级别和其他事件等同对待，但Upstart系统中的很多任务配置文件涉及运行级别。

系统启动过程中有一个关键点，就是当所有文件系统都挂载完毕，大部分重要系统都初始化后。此时系统准备启动更高级别的系统服务，如图形显示管理和数据库服务。此时产生一个`runlevel`事件以做标记，你也可以配置Upstart产生其他事件。判断哪些服务作为Upstart任务启动、哪些作为System V链接池（参见6.6.2节）启动不是一件容易的事。比如你的运行级别是2，则/etc/rc2.d中的任务很有可能是以System V兼容模式运行。

注解　/etc/init.d文件中的伪脚本比较让人头疼。对于Upstart的service任务，/etc/init.d中可能有一个与之对应的System V脚本，但是它除了表示该服务已经被转换为Upstart任务以外，并没有其他作用。也没有到System V链接目录脚本的符号链接。如果你看到伪脚本，你可以获得Upstart任务名，然后使用`initctl`来操控该任务。

6.6　System V init

Linux上的System V init实现要追溯到Linux的早期版本，它根本目的是为了为系统提供合理的启动顺序，支持不同的运行级别。虽然现在System V已经不太常见，不过在Red Hat Enterprise Linux和一些路由器和电话的Linux嵌入系统中还是能够看到System V init。

典型的System V init安装包含两个主要组件：一个核心配置文件和一组启动脚本以及符号链接集。配置文件/etc/inittab是核心。如果你系统中有System V init的话，你可以从中看到如下内容：

```
id:5:initdefault:
```

这表示运行级别默认为5。

inittab中的内容都有如下格式，四列内容使用分号隔开，分别是：

- ❑ 唯一标识符（一串短字符，本例中为id）；
- ❑ 运行级别值（一个或多个）；
- ❑ init执行的操作（本例中是将运行级别设置为5）；
- ❑ 执行的命令（可选项）。

下面一行内容告诉我们命令如何运行：

```
l5:5:wait:/etc/rc.d/rc 5
```

这行内容很重要，它触发大部分的系统配置和服务。wait操作决定System V init何时和怎样运行命令。进入运行级别5时运行一次/etc/rc.d/rc 5，然后一直等待命令执行完毕。rc 5命令运行/etc/rc5.d中所有以数字开头的命令（按数字的顺序）。

除了initdefault和wait之外，下面是其他inittab的常见操作。

respawn

respawn让init在其后的命令结束执行后，再次运行。在inittab文件中你有可能会看到以下内容：

```
1:2345:respawn:/sbin/mingetty tty1
```

getty程序提供登录提示符。上面的命令是针对第一个虚拟控制台（/dev/tty1）的，当你按ALT-F1或者CTRL-ALT-F1时能够看到（参考3.4.4节）。respawn在你退出系统后重新显示登录提示符。

ctrlaltdel

ctrlaltdel是控制当你在虚拟控制台中按CTRL-ALT-DEL键时系统采取的操作。在大部分系统中，这是重启命令，它执行shutdown命令（我们将在6.7节介绍）。

sysinit

sysinit是init在启动过程中，且在进入运行级别之前执行的第一个操作。

注解 请在inittab(5)帮助手册中查看更多的操作。

6.6.1 System V init启动命令顺序

现在你可以来了解一下，在你登录系统之前System V init是怎样启动系统服务的。之前我们介绍过下列命令行：

```
l5:5:wait:/etc/rc.d/rc 5
```

它只是一行简单的指令，但实际触发了很多其他程序。rc是运行命令（run command）的简写，我们会在许多脚本、程序和服务中使用到它。那么运行的命令在哪里呢？

该行中的5代表运行级别5。运行的命令多半是在/etc/rc.d/rc5.d或者/etc/rc5.d中。（运行级别1使用rc1.d，运行级别2使用rc2.d，以此类推。）你可能会在rc5.d目录下找到以下内容：

```
S10sysklogd     S20ppp          S99gpm
S12kerneld      S25netstd_nfs   S99httpd
S15netstd_init  S30netstd_misc  S99rmnologin
S18netbase      S45pcmcia       S99sshd
S20acct         S89atd
S20logoutd      S89cron
```

rc 5命令通过执行下面的命令来运行rc5.d目录下的程序：

```
S10sysklogd start
S12kerneld start
S15netstd_init start
S18netbase start
--snip--
S99sshd start
```

请注意每一行中的start参数。命令名中的大写S表示命令应该在start模式中运行，数字00~99决定了rc启动命令的顺序。rc*.d命令通常是命令行脚本，用来启动/sbin或者/usr/sbin中的程序。

一般情况下，你可以使用less或其他命令来查看脚本文件内容，进而了解各种命令的功能。

注解　有一些rc*.d目录中包含以K（代表kill，或者stop模式）开头的命令。此时rc使用参数stop而非start运行命令。K开头的命令通常在关闭系统的运行级别中。

你也可以手动运行这些命令。不过通常你是通过init.d目录而非rc*.d来运行，我们随后就讲到。

6.6.2　System V init链接池

rc*.d目录实际上包含的是符号链接，指向init.d目录中的文件。如果想运行、添加、删除或者更改rc*.d目录中的服务，你需要了解这些符号链接。下面是rc5.d目录的内容示例：

```
lrwxrwxrwx . . . S10sysklogd -> ../init.d/sysklogd
lrwxrwxrwx . . . S12kerneld -> ../init.d/kerneld
lrwxrwxrwx . . . S15netstd_init -> ../init.d/netstd_init
lrwxrwxrwx . . . S18netbase -> ../init.d/netbase
--snip--
lrwxrwxrwx . . . S99httpd -> ../init.d/httpd
--snip--
```

像上例所示这样，子目录中所包含的大量符号链接，我们称之为链接池（Link Farm）。有了这些链接，Linux可以对不同的运行级别使用相同的启动脚本。虽然不需要严格遵循，但这种方法确实更简洁。

启动和停止服务

如果要手动启动和停止服务，可以使用init.d目录中的脚本。比如我们可以运行init.d/httpd start来启动httpd Web服务。类似地，使用stop参数（如httpd stop）来关闭服务。

更改启动顺序

在System V init中更改启动顺序是通过更改链接池来完成的。最常见的更改是禁止init.d目录中的某个命令在某个特定的运行级别中运行。在进行此操作时需谨慎。因为假如你删除了某个rc*.d目录中的一个符号链接，将来你想恢复它的时候，你可能已经忘记了它的链接名。所以一个比较好的办法是在链接名前加下划线（_），如：

```
# mv S99httpd _S99httpd
```

这一指令使得rc忽略_S99httpd，因为文件名不以S或K开头，同时又保留了原始的链接名。

如果要添加服务，我们可以在init.d目录中创建一个脚本文件，然后在相应的rc*.d目录中创建指向它的符号链接。最简单的办法是在init.d目录中复制和修改你熟悉的脚本（更多命令行脚本的内容参见第11章）。

在添加服务的时候，需要为其设置适当的启动顺序。如果服务启动过早，可能会导致失败，因为它依赖的其他服务可能还没有就绪。对于那些不重要的服务，大多数系统管理员会为它们设置90以后的序号，以便让系统服务首先启动。

6.6.3 run-parts

System V init运行init.d脚本的机制在很多Linux系统中被广泛应用，甚至包括那些没有System V init的系统。其中有一个工具我们称为run-parts，它能够按照特定顺序运行指定目录中的所有可执行程序。这好比是用户使用ls命令列出目录中的程序，然后逐一运行。

run-parts默认运行目录中的所有可执行程序，也有可选项用来指定执行或忽略某些特定程序。在一些Linux系统中，你不太需要控制这些程序如何运行。如Fedora就只包含一个很简单的run-parts版本。

其他一些Linux系统，如Debian和Ubuntu则包含一个复杂一些的run-parts程序。其功能包括使用正则表达式来选择运行程序（例如，使用S[0-9]{2}来运行/etc/init.d运行级别目录中的所有启动脚本），并且还能够向这些程序传递参数。这些特性能够让我们使用一条简单的命令来完成System V运行级别的启动和停止。

关于run-parts的细节你不需要知道太多，很多人甚至不知道有run-parts这么个东西。你只需要知道它能够运行一个目录中的所有程序，在脚本中时不时会出现即可。

6.6.4 System V init控制

有些时候，你需要手动干预一下init，以便它能够切换运行级别，或者重新加载配置信息，甚至关闭系统。你可以使用telinit来操纵System V init。例如，使用以下命令切换到运行级别3：

```
# telinit 3
```

运行级别切换时，init会试图终止所有新运行级别的inittab文件中没有包括的进程，所以需要小心操作。

如果你需要添加或者删除任务，或者更改inittab文件，你需要使用telinit命令让init重新加载配置信息：

```
# telinit q
```

可以使用telinit s命令切换到单用户模式（见6.9节）。

6.7　关闭系统

init控制系统的启动和关闭。关闭系统的命令在所有init版本中都是一样的。关闭Linux系统最好的方式是使用shutdown命令。

shutdown命令有两种使用方法，一是使用-h可选项关闭系统，并且使其一直保持关闭状态。下面的命令能够立即关闭系统：

```
# shutdown -h now
```

在大部分系统中，-h代表切断机器电源。另外还可以使用-r来重启系统。

系统的关闭过程会持续几秒钟，在此过程中请不要重置或切断电源。

上例中的now是时间，是一个必须的参数。设置时间的方法有很多种。例如，如果你想让系统在将来某一时间关闭，可以使用+n。n以分钟为单位，系统会在n分钟后执行关闭命令。（可以在shutdown(8)帮助手册中查看更多相关选项。）

下面的命令在10分钟后重启系统：

```
# shutdown -r +10
```

Linux会在shutdown运行时通知已经登录系统的用户，不过也仅此而已。如果你将时间参数设置为now以外的值，shutdown命令会创建一个文件/etc/nologin。这个文件存在时，系统会禁止超级用户外的任何用户登录。

系统关闭时间到时，shutdown命令通知init开始关闭进程。在Systemd中，这意味着激活关闭单元；在Upstart中，这意味着产生关闭事件；在System V init中，这意味着将运行级别设置为0或6。无论哪个系统，关闭过程都大致如下所示。

(1) init通知所有进程安全关闭。

(2) 如果某个进程没有及时响应，init会先使用TERM信号尝试强行终止它。

(3) 如果TERM信号无效，init会使用KILL信号。

(4) 锁定系统文件，并且进行其他关闭准备工作。

(5) 系统卸载root以外的所有文件系统。

(6) 系统以只读模式重新挂载root文件系统。

(7) 系统将所有缓冲区中的数据通过sync程序写到文件系统。

(8) 最后一步是使用reboot(2)系统调用通知内核重启或者停止。这一步骤是由init或者其他辅助程序如reboot、halt或者poweroff来完成。

reboot和halt程序因它们被调用的方式不同而行为各异，有时还会带来一些困扰。默认情况下，它们使用参数-r或者-h来调用shutdown。但如果系统已经处于halt或者reboot的运行级别，则程序会通知内核立即关闭自己。如果你想不计后果快速关闭系统，可以使用-f（force）选项。

6.8 initramfs

Linux启动过程很简单。但是其中的一个组件总是让人一头雾水，那就是initramfs，或称为*初始RAM文件系统*。可以把它看作是一个用户空间的楔子，在用户空间启动前出现。不过首先我们来看看它是用来做什么的。

问题还得从种类各异的存储硬件说起。不知道你是否还记得，Linux内核从磁盘读取数据时不直接与BIOS和EFI接口通信。为了挂载root文件系统，它需要底层的驱动程序支持。如果root文件系统存放在一个连接到第三方控制器的磁盘阵列（RAID）上，内核首先就需要这个控制器的驱动程序。因为存储控制器的驱动程序种类繁多，内核不可能把它们都包含进来，所以很多驱动程序都以可加载模块的方式出现。可加载模块是以文件形式来存放，如果内核一开始没有挂载文件系统的话，它就无法加载需要的这些驱动模块了。

解决的办法是将一小部分内核驱动模块和工具打包为一个文档。引导装载程序在内核运行前将该文档载入内存。内核在启动时将文档内容读入一个临时的initramfs，然后挂载到/上，将用户模式切换给initramfs上的init，然后使用initramfs中的工具让内核加载root文件系统需要的驱动模块。最后，这些工具挂载真正的root文件系统，启动真正的init。

initramfs的具体实现各有不同，并且还在不断演进。在一些系统中，initramfs的init就是一个简单的命令行脚本，通过udevd来加载启动程序，然后挂载真正的root并在其上执行init。在使用systemd的系统中，你能在其中看到整个的systemd安装，但没有单元配置文件，只有一些udevd配置文件。

initramfs的一个始终未变的特性是你可以在不需要时跳过它。也就是说，如果内核已经有了所有它需要的用来挂载根文件系统的驱动程序，你就可以在你的引导装载程序配置中跳过initramfs。跳过该过程能够缩短几秒钟的启动时间。你可以自己尝试在GRUB菜单编辑器中删除initrd行。（最好不要使用GRUB配置文件来做实验，一旦出错很难恢复。）目前来说，initramfs还是需要的，因为大多数的Linux内核并不包含诸如通过UUID挂载这些特性。

initramfs只是通过gzip压缩的cpio归档文件（见帮助手册cpio(1)）。你可以先从引导装载程序配置中找到该文件（例如使用grep在grub.cfg文件中查找initrd），然后使用cpio将归档文件的内容释放到一个临时目录来查看其内容，如下例所示：

```
$ mkdir /tmp/myinitrd
$ cd /tmp/myinitrd
$ zcat /boot/initrd.img-3.2.0-34 | cpio -i --no-absolute-filenames
--snip--
```

其中有一处地方值得一提，就是init末尾的“pivot”部分。它负责清除临时文件系统中的内容，以节省内存空间并切换到真正的root。

创建initramfs的过程很复杂，不过通常我们不需要自己动手。有很多工具可以供我们使用，Linux系统中通常都会自带，如dracut和mkinitramfs是最为常用的两个。

> **注解**　initramfs是指使用cpio归档文件作为临时文件系统。它的一个较老的版本叫作initrd（即初始RAM磁盘），它使用磁盘映像文件作为临时文件系统。cpio归档文件的维护更加简单，不过很多时候initrd也用来代指使用cpio的initramfs。如上例所示，文件名和配置文件中都仍然包含initrd。

6.9　紧急启动和单用户模式

当系统出现问题时，首先采取的措施通常是使用系统安装映像来启动系统，或者使用SystemRescueCd这样可以保存到移动存储设备上的恢复映像。系统修复的任务大致包括以下几方面：

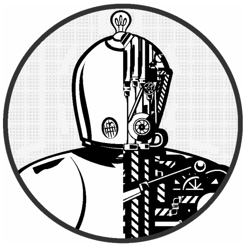
- 系统崩溃后，检查文件系统；
- 重置系统管理员密码；
- 修复关键的系统文件，如/etc/fstab和/etc/passwd；
- 系统崩溃后，借助备份数据恢复系统。

除上述措施外，快速启动的另一个可选途径是启用单用户模式。它将系统启动到root命令行，而不是完整启动所有的服务。在System V init中，运行级别1通常是单用户模式。你也可以在引导装载程序中使用-s参数来进入此模式，只不过可能需要输入root密码。

单用户模式的最大问题是它提供的服务有限，如网络、图形界面和终端通常都不可用。所以我们在系统恢复时通常优先考虑系统安装映像。

第 7 章

系统配置：日志、系统时间、批处理任务和用户



当你第一次看到/etc目录时，可能会感觉信息量太大。固然，其中的大多数文件或多或少都对系统运行有影响，但只有少数文件起非常关键的作用。

本章我们介绍一些系统组件，它们使得用户级工具（参见第2章）能够访问系统的基础设施（参见第4章）。我们将着重介绍以下内容：

- 系统库为获得服务和用户信息而访问的配置文件；
- 系统启动时运行的服务程序（有时称为守护进程）；
- 用来更改服务程序和配置文件的配置工具；
- 系统管理工具。

本章不涉及网络相关的内容，因为那是一个相对独立的部分，我们将在第9章介绍。

7.1 /etc 目录结构

Linux系统的大部分系统配置文件都存放在/etc目录中。按照惯例，每个程序在这里都有一个或多个配置文件。因为Unix系统的程序数目很多，所以/etc目录也会越来越庞大。

这样带来了两个问题：不仅很难找到要找的配置文件，而且维护起来也不方便。比如，要更改系统的日志配置，你需要编辑/etc/syslog.conf文件。但是你的更改可能会被随后的系统升级覆盖掉。

目前比较常见的方式是将系统配置文件放到/etc下的子目录，像我们介绍过的启动目录一样（Upstart的是/etc/init，systemd的是/etc/systemd）。虽然/etc目录下仍然会有一些零散的配置文件，

如果你运行`ls -F /etc`查看的话，你会发现大部分配置文件都放到了子目录中。

为了解决配置文件被覆盖的问题，你可以将定制的配置放到子目录里的其他文件中，如/etc/grub.d。

/etc中到底有哪些类型的配置文件呢？基本规律是针对系统的可定制的配置文件在/etc下，如用户信息（/etc/passwd）和网络配置（/etc/network）。然而，与应用程序细节相关的文件不在/etc中，如系统用户界面的默认配置。你会发现那些不可定制的系统配置文件存放在了其他地方，比如预先打包的系统单元文件在/usr/lib/systemd中。

我们已经介绍过一些启动相关的配置文件。下面让我们来看一个具体的系统服务及其配置文件。

7.2　系统日志

大多数系统程序将它们的日志信息输出到syslog服务。传统的`syslogd`守护进程等待消息的到来，并根据它们的类型将它们输出到文件、屏幕、用户或其他地方，有的干脆忽略。

7.2.1　系统日志

系统日志是系统中最重要的部分之一。如果系统出现你不清楚的错误，查看系统日志文件是第一选择。以下是日志文件示例：

```
Aug 19 17:59:48 duplex sshd[484]: Server listening on 0.0.0.0 port 22.
```

大多数Linux系统使用的是`syslogd`的一个新版本，叫作`rsyslogd`。它的功能不仅仅限于记录日志信息。比如，你还可以让它加载一个将日志信息写到数据库的模块。不过最简单的方式还是从/var/log目录开始。你看看其中的那些日志文件后，就能够了解它们来自哪里。

/var/log目录中很多文件都不是由系统日志来维护的。要知道哪些日志属于`rsyslogd`，需要查看它们的配置文件。

7.2.2　配置文件

`rsyslog`的基础配置文件是/etc/rsyslog.conf，但你还会在其他地方（如/etc/rsyslog.d）发现另外一些配置文件。其内容包含传统的规则和`rsyslog`扩展。其中一条规则是任何以字符$开头的都是扩展。

传统的规则包括一个选择符（selector）和一个操作（action），分别代表从哪里获得日志和将它们写到哪里，如下例所示：

例7-1　syslog规则

```
kern.*                  /dev/console
*.info;authpriv.none❶   /var/log/messages
```

```
authpriv.*                  /var/log/secure,root
mail.*                      /var/log/maillog
cron.*                      /var/log/cron
*.emerg                     *❷
local7.*                    /var/log/boot.log
```

左边是选择符，表示要为哪种信息类型记录日志。右边是操作列表，表示要将日志写到哪里。例7-1中大部分的操作都是将日志写入文件，也有一些例外。例如：/dev/console表示系统控制台的一个特殊设备，root表示如果root用户登录的话，将消息发送给他，*代表发送消息给系统中的所有用户。你还可以使用@host将消息发送给网络上的其他主机。

设施和优先级

选择符用来匹配日志信息的设施和优先级。设施是指消息的大致分类。（在rsyslog.conf(5)帮助手册中查看完整的设施列表。）

设施的功能很容易通过它们的名称得知。例如，例7-1中的配置文件从kern、authpriv、mail、cron和local7这些设施中抓取日志信息。❷处的*号是一个通配符，表示从所有设施中获得输出。

设施后的点号（.）后面是优先级，由低到高分别是：debug、info、notice、warning、err、crit、alert或emerge。

7

> **注解**　要在rsyslog.conf中将日志消息从设施中排除，可以使用none作为优先级，如例7-1中❶所示。

为选择符设置了优先级之后，rsyslogd将该优先级及其以上优先级的消息发送到指定目的地。也就是说，例7-1中❶处的*.info将抓取大部分日志消息并将它们写到/var/log/messages，因为info是一个相对较低的优先级。

扩展语法

之前提到过，rsyslogd的语法扩展了传统的syslogd语法，它们通常以$开头，我们称之为指令。一个比较常用的扩展是让你加载其他的配置文件。rsyslog.conf中就包含这样的指令，让rsyslogd加载/etc/rsyslog.d目录中的所有.conf文件。

```
$IncludeConfig /etc/rsyslog.d/*.conf
```

大部分的指令都很好理解，比如以下涉及用户和权限的指令：

```
$FileOwner syslog
$FileGroup adm
$FileCreateMode 0640
$DirCreateMode 0755
$Umask 0022
```

注解 还有一些rsyslogd配置文件扩展定义了输出模板和频道，你可以查看rsyslogd(5)完整的帮助手册，不过它的Web文档更全面一些。

故障排除

测试系统日志最简单的方法之一是使用logger命令手动发送日志消息，如下所示：

```
$ logger -p daemon.info something bad just happened
```

rsyslogd不容易出错。出现问题大都是因为配置文件没有正确配置设施或优先级，因而没有获得想要抓取的日志信息，或者是由于磁盘空间不足。大多数系统会使用logrotate或者类似工具来自动清除/var/log中的文件，不过如果在短时间写入大量日志，还是会出现系统负载增加、磁盘空间用尽的情况。

注解 rsyslogd抓取的日志不仅仅来自于系统各组件。我们在第6章介绍过systemd和Upstart抓取的启动日志消息，除此之外还有很多其他来源，比如Apache Web服务器，通常它有自己的存取和错误日志。你可以查看服务器配置来获得那些日志。

日志的过去和未来

syslog服务在不断演进。曾经出现过一个叫klogd的守护进程，负责为syslogd截获内核的日志消息，这些日志可以使用dmesg命令查看。后来该功能被并入到了rsyslogd中。

毋庸置疑的是，Linux的系统日志功能会随着时间而改变。Unix系统日志从来就没有形成过真正的标准，不过这种情况正在慢慢改变。

7.3　用户管理文件

Unix系统支持多用户。用户对于内核而言只是一些数字（用户ID），因为用户名比数字容易记忆，所以用户一般都是用用户名（或登录名）而非用户ID来管理系统。用户名只存在于用户空间，使用到用户名的应用程序在和内核通信时，通常需要将用户名映射为用户ID。

7.3.1　/etc/passwd文件

文本文件/etc/passwd中包含一一对应的用户名和用户ID。如下所示：

例7-2　/etc/passwd中的用户列表

```
root:x:0:0:Superuser:/root:/bin/sh
daemon:*:1:1:daemon:/usr/sbin:/bin/sh
bin:*:2:2:bin:/bin:/bin/sh
sys:*:3:3:sys:/dev:/bin/sh
nobody:*:65534:65534:nobody:/home:/bin/false
```

```
juser:x:3119:1000:J. Random User:/home/juser:/bin/bash
beazley:x:143:1000:David Beazley:/home/beazley:/bin/bash
```

每一行代表一个用户，一共有7列，用冒号:分隔。这7列所代表的内容如下所示。

- ❑ 登录名。
- ❑ 经过加密的用户密码。大部分Linux系统都不在passwd文件中存放实际的用户密码，而是将密码存放在shadow文件中（见7.3.3节）。shadow文件的格式和passwd类似，不过普通用户没有访问权限。passwd和shadow文件中的第2列是经过加密的密码，是一些像d1CVEWiB/oppc这样的字符，读起来很费劲。（Unix从不明文存储密码。）

  第2列中的x代表加密过的密码存放在shadow文件中。*代表用户不能登录，如果为空（像::这样），则表示登录不需要密码。（绝对不要将普通用户的该列设置为空。）
- ❑ 用户ID。它是用户在内核中的标识。同一个用户ID可以出现在两行中，不过这样做比较容易产生混淆，程序在处理时也需要将它们合并起来。用户ID必须唯一。
- ❑ 用户组ID。它是/etc/group文件中的某个ID号。用户组定义了文件权限及其他。该列也称为用户的基本组。
- ❑ 用户的真实名称（通常称为GECOS列）。有时候其中会有逗号，用来分隔房间和电话号码。
- ❑ 用户的root目录。
- ❑ 用户使用的命令行，即用户运行终端的程序。

图7-1标识出了例7-2中条目的各列。

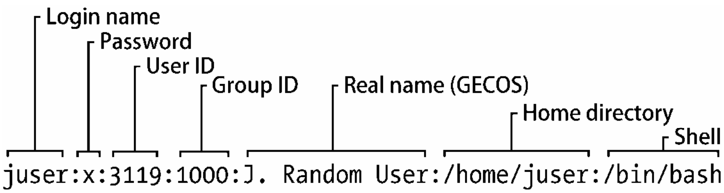


图7-1 passwd文件中的条目

/etc/passwd文件有严格的语法规则，不允许注释和空行。

注解 用户在/etc/passwd中的对应行及其root目录统称为用户账号。

7.3.2 特殊用户

在/etc/passwd中有一些特殊用户。其中，超级用户的UID和GID固定为0，如例7-2所示。有一些用户如守护进程用户没有登录权限。nobody用户的权限最小。一些进程在nobody用户名下运行，因为它没有任何写入权限。

无法登录的用户我们称为伪用户。虽然无法登录系统，但是系统可以使用它们来运行一些进程。创建像nobody用户这样的伪用户，目的是为了安全考虑。

7.3.3 /etc/shadow文件

Linux中的影子密码文件（/etc/shadow）包含用户验证信息以及经过加密的密码和密码过期日期，这些都和/etc/passwd文件中的用户相对应。

影子文件为密码存储提供了一种更灵活（同时也更安全）的方法。它包括了一些程序库和工具，后来很快被PAM替代（参考7.10节）。PAM并没有为Linux引进一套全新的文件，而是使用/etc/shadow文件，但像/etc/login.defs这样对应的配置文件并不使用它。

7.3.4 用户和密码管理

普通用户使用`passwd`命令来更改密码。默认状态下，`passwd`可以更改用户密码，但你还可以使用`-f`选项来更改用户名，用`-s`选项来更改shell（/etc/shells中有shell列表）。（你还可以使用`chfn`和`chsh`来更改用户名和shell。）`passwd`命令是一个suid-root程序，只有超级用户能够编辑/etc/passwd文件。

使用超级用户来更改/etc/passwd

由于/etc/passwd是纯文本文件，因此超级用户可以使用任何文本编辑器来编辑它。要添加用户，只需加上恰当的命令行并为用户创建一个root目录即可。要删除用户则反之。不过通常我们使用`vipw`来编辑/etc/passwd文件，它更为安全，会在你编辑时备份和锁定文件。你还可以使用`vipw -s`来编辑/etc/shadow文件（但你可能永远不需要用到）。

很多人不愿意直接编辑passwd文件，因为很容易把文件搞乱。使用另外的终端命令或者GUI会更为方便和安全。比如，可以使用超级用户运行`passwd user`来设置用户密码。`adduser`和`userdel`可以添加和删除用户。

7.3.5 用户组

用户组可以将文件访问权设定给某些用户，而使其他用户无权访问。你可以为某组用户设置读写位，从而排除其他的用户。在多名用户共享一台主机的时候，用户组很有用。然而现在我们很少在主机上共享文件了。

/etc/group文件中包含了用户组ID（类似/etc/passwd文件中的ID），如例7-3所示。

例7-3 /etc/group文件样例

```
root:*:0:juser
daemon:*:1:
bin:*:2:
sys:*:3:
adm:*:4:
disk:*:6:juser,beazley
nogroup:*:65534:
user:*:1000:
```

和/etc/passwd文件一样，/etc/group中的每一行有多列，由冒号分隔。每一列所代表的内容如下所示。

- 用户组名：运行如`ls -l`这样的命令时可以看到。
- 用户组密码：很少也不该被使用（使用sudo替代）。可以设置为*或者其他默认值。
- 用户组ID：必须是一个唯一的数字。用户组ID出现在/etc/passwd文件的用户组列中。
- 属于该组的用户列表：该列是可选项，passwd文件中的用户组ID列也定义了用户属于哪个用户组。

图7-2显示了用户组文件中的各列：

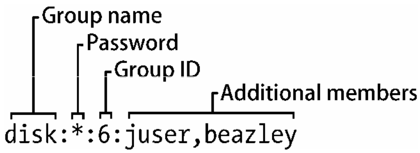


图7-2　group文件中的条目

你可以使用`group`命令来查看你所属的用户组。

注解　Linux通常会为每个新加入的用户创建一个新的用户组，用户组名和用户名相同。

7.4　getty 和 login

`getty`连接到终端并且在其上显示登录提示符。大多数Linux系统中的`getty`程序很简单，仅仅是在虚拟终端上显示登录提示符。它可以用在管道命令中，如下所示：

```
$ ps ao args | grep getty
/sbin/getty 38400 tty1
```

本例中，38400是波特率。有些`getty`不需要该设置。（虚拟终端会忽略此设置，这只是为了和连接串行口的那些程序兼容。）

输入用户名后，`getty`调用`login`程序提示你输入密码。如果输入的密码正确，`login`会调用你的shell（使用`exec()`）。否则你会得到“登录错误”提示信息。

现在你了解了`getty`和`login`，但你可能永远都不需要配置和更改它们。实际上，你使用它们的机会可能不多，因为现在用户大都通过图形界面（如`gdm`）或者远程登录（如SSH），这些都用不着`getty`和`login`。登录程序的实际验证工作大多是由PAM来完成的（参考7.10节）。

7.5　设置时间

Unix系统的运行依赖精确的计时，而内核则负责维护系统时钟。你可以使用`date`命令来查看，

还可以用它设置时间，不过并不推荐这样做，因为设置的时间有可能不精准，而你的系统时间应该尽可能精准。

计算机硬件有一个使用电池的实时时钟（Real-time Clock，以下简称RTC）。RTC并不是最精准的，但是聊胜于无。内核通常在启动时使用RTC来设置时间。你可以使用hwclock命令将系统时间重新设置为硬件系统的当前时间。最好将你的硬件时钟设置为通用协调时间（Universal Coordinated Time，以下简称UTC），这样可以避免不同时区和夏令时带来的问题。你可以使用以下命令将内核的UTC时钟设置为RTC：

```
# hwclock --hctosys --utc
```

不过内核在计时方面还不如RTC，因为Unix系统启动一次经常持续运行数月甚至数年，所以容易产生时间误差（time drift）。这个误差是指系统时间的实际时间（通常由原子时钟等精确时钟来定义）之差。

不要试图使用hwclock来修复时间误差，因为这会影响那些基于时间的系统事件。你可以运行adjtimex来更新系统时钟，不过最好的办法是使用守护进程来使你的系统时间和网络上的时间保持同步（参见7.5.2节）。

7.5.1　内核时间和时区

内核将当前的系统时间显示为以秒为单位的一串数字，自UTC时间1970年1月1日12:00时起开始。你可以使用以下命令来查看：

```
$ date +%s
```

为了保证易读性，用户空间程序会将这组数字转换为本地时间，并且将夏令时和其他因素（比如印第安纳州时间）都考虑在内。文件/etc/localtime（二进制文件）用来控制本地时区。

时区信息在/usr/share/zoneinfo目录中，其中包含了时区及其别名等信息。如果要手动设置时区，可以将/usr/share/zoneinfo中的某个文件复制到/etc/localtime（或者创建一个符号链接）中，或者使用系统自带的时区工具。（你可以使用tzselect命令寻找时区文件。）

如果要为shell会话设置时区，可以将TZ环境变量设置为/usr/share/zoneinfo中的某个文件名，如下所示：

```
$ export TZ=US/Central
$ date
```

和其他环境变量一样，你也可以只为某条命令的执行持续时间设置时区，如下所示：

```
$ TZ=US/Central date
```

7.5.2 网络时间

如果你的主机连接到互联网，你可以运行网络时间协议（Network Time Protocol，以下简称NTP）守护进程，借助远程服务器来更新时间。很多Linux系统自带NTP守护进程，但是不一定默认开启。你可以安装ntpd包来运行它。

如果你想手动配置，可以参考NTP网站 http://www.ntp.org。如果你想偷点懒也没关系，下面是简要的步骤：

(1) 从你的ISP或者ntp.org获得离你最近的NTP服务器；

(2) 将该服务器加入/etc/ntpd.conf文件；

(3) 在启动时运行`ntpdate server`；

(4) 启动时，在`ntpdate`命令之后运行`ntpd`。

如果你的主机没有连接互联网，你可以使用`chronyd`守护进程在离线状态下维护系统时间。

在系统重启时，你还可以根据网络时间来设置系统的硬件时钟，为的是帮助系统保持时间的一致性。（很多Linux系统会自动这样做。）使用`ntpdate`（或者`ntpd`）从网络设置系统时间，然后运行我们在前面介绍过的命令：

```
# hwclock --systohc --utc
```

7.6 使用 cron 来调度日常任务

Unix的cron（意思是定时）服务能够按照日程安排来重复运行程序。cron对多数富有经验的系统管理员来说非常重要，因为它可以完成很多自动化的系统维护工作。比如，它运行日志文件的替换工具来确保旧的日志文件被删除以腾出磁盘空间。建议你掌握cron的使用方法，这对你会很有帮助。

你可以使用cron在任何时间运行任何程序。通过cron运行的程序我们称为*定时任务*。要添加一个定时任务，可以在crontab（意为定时任务）文件中加入一行，通常是通过执行`crontab`命令来完成。例如，你若想将`/home/juser/bin/spmake`命令安排在每天9:15AM运行，可以加入以下一行：

```
15 09 * * * /home/juser/bin/spmake
```

最开始的5列用空格分隔，设定任务运行的时间（参见图7-3），它们的含义如下所示。

- 分（0 ~ 59）：上例中是`15`。
- 时（0 ~ 23）：上例中是`09`。
- 天（1 ~ 31）。
- 月（1 ~ 12）。
- 星期（0 ~ 7）：0和7代表周日。

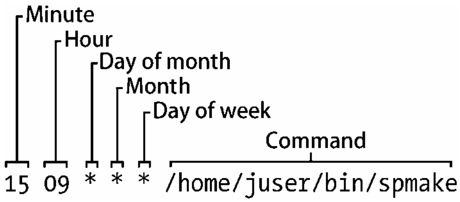


图7-3　crontab文件中的条目

任意位置出现的星号表示匹配所有值。上例中spmake每天都运行，因为天、月、星期等列的值都是星号，所以cron就把它解读为“每月每周的每一天都要运行这个任务”。

如果只想在每个月的14号运行spmake，可以使用下面这行设置：

```
15 09 14 * * /home/juser/bin/spmake
```

每一列可以有多个值。例如，要在每月5号和14号运行程序，可以将第三列设置为5,14：

```
15 09 5,14 * * /home/juser/bin/spmake
```

注解　如果cron任务产生标准输出、错误或者非正常退出，你会收到一封邮件通知。如果你觉得邮件太麻烦，可以将输出结果重定向到/dev/null或者日志文件中。

帮助手册crontab(5)为我们提供了有关crontab的详细信息。

7.6.1　安装crontab文件

每个用户都可以有自己的crontab文件，所以系统中经常会有很多个crontab，通常保存在/var/spool/cron/crontabs目录中。普通用户对该目录没有写权限，crontab命令负责安装、查看、编辑和删除用户的crontab。

安装crontab最简便的方法是将crontab条目放入一个文件（如file），然后运行crontab file命令将file文件安装为你的crontab。crontab命令会检查文件的格式，确保没有错误。你可以使用crontab -l列出你的cron任务。使用crontab -r删除crontab文件。

然而在你第一次创建了crontab文件后，后续使用临时文件来进行更改会比较麻烦。你可以使用crontab -e命令来更改并安装你的crontab。如果有错误，crontab命令会提示你错误出在哪里，并询问是否重新编辑。

7.6.2　系统crontab文件

Linux系统通常使用/etc/crontab文件来安排系统任务的运行，而不是使用超级用户的crontab。不要使用crontab命令来编辑该文件，因为它有一个额外的列来设置运行任务的用户。例如下面

这一行设置，任务在6：42AM由root（❶处）用户运行：

```
42 6 * * * root❶ /usr/local/bin/cleansystem > /dev/null 2>&1
```

注解 一些系统将系统crontab文件存放在/etc/cron.d目录中。它们的文件名也许不同，不过内容格式和/etc/crontab一样。

7.6.3 cron的未来

cron工具是Linux系统中历史最长的组件之一，大约有几十年了，甚至在Linux出现之前就已经存在。它的文件格式一直以来基本上没有改变。由于它实在是太过于老旧，人们正在想办法替换它。

它的可行的替代者实际上是新版本的init的一部分：对于systemd来说是计时器单元；对于Upstart来说是重复产生事件来触发任务。总之它们都能够以任何用户的名义运行任务，并且拥有诸如定制日志这样的便利之处。

但实际上目前的systemd和Upstart都还不具备cron的全部功能。而且，就算它们具备所有功能，也还要考虑向后兼容性，要能够支持那些依赖于cron的系统。从这个意义上说，cron还不会这么快被替代。

7.7 使用 at 进行一次性任务调度

要想在将来的某一时刻一次性运行任务，如果不使用cron的话，可以使用at服务。例如要在10:30PM运行myjob，可以使用以下命令：

```
$ at 22:30
at> myjob
```

使用CTRL-D结束输入。（at从标准输入读取命令。）

要检查任务是否已经被设定，可以使用atq。要删除任务，使用atrm。你还可以使用DD.MM.YY这样的日期格式将任务设置为将来某一时刻运行，如：at 22:30 30.09.15。

关于at命令差不多就这些内容。虽然它不太常用，不过在某些场景下比较有用，比如你想让系统在将来某个时刻关闭的时候。

7.8 了解用户 ID 和用户切换

我们已经介绍过，sudo和su这样的setuid程序允许你切换用户，login这样的系统组件负责控制用户访问。你或许想了解它们的工作原理，以及内核在用户切换中所起的作用。

更改用户ID有两种方式，均由内核负责完成。第一种是运行setuid程序，我们在2.17节介绍过。第二种是通过`setuid()`系统调用，该系统调用有很多不同版本，用来处理和进程关联的所有用户ID，我们将在7.8.1节详细介绍。

内核负责为进程制定规则，规定哪些能做，哪些不能做。下面是三个基本规则：

- 以root（userid 0）身份运行的进程可以调用`setuid()`来切换为任何其他用户；
- 没有以root身份运行的进程在调用`setuid()`时有一些限制，大多数情况下不能调用`setuid()`；
- 任何进程，只要有足够的文件访问权限，都可以运行setuid程序。

> **注解**　用户切换并不涉及用户名和用户密码。这是用户空间的概念，我们在7.3.1节介绍过并将在7.9.1节中详细介绍。

进程归属、有效UID、实际UID和已保存UID

到目前为止，本书所讲的有关用户ID的内容都是简化过的。实际上每个进程都有超过一个用户ID。我们提到过的*有效UID*（effective user ID，euid）是用来设定某一进程的访问权限。另外还有一个UID是*实际UID*（real user ID，ruid），即实际启动进程的UID。当你运行setuid程序时，Linux将有效UID设置为程序文件的拥有者，同时将实际UID设置为你的UID。

在很多现代系统中，有效UID和实际UID之间的区别很模糊，以至于很多文档中有关进程归属的内容都是不正确的。

我们可以将有效UID看作*执行者*，将实际UID看作*所有者*。实际UID是可以与进程进行交互的用户，可以终止进程，向进程发送信号。例如，如果用户A以用户B的名义启动了一个新进程（基于setuid权限），用户A仍然是该进程的所有者，并且可以终止该进程。

在Linux系统中，大多数进程的有效UID和实际UID是相同的。`ps`和其他系统诊断命令默认显示有效UID。要想查看你系统上的有效UID和实际UID，可以使用以下命令，但你会发现，你的系统中的所有进程几乎拥有相同的有效UID和实际UID。

```
$ ps -eo pid,euser,ruser,comm
```

如果想要两者有不同的值，可以为`sleep`命令创建一个setuid副本，运行一段时间，然后在其结束前使用ps命令在另一个终端窗口查看它的信息。

除了有效UID和实际UID外，还有一个*已保存UID*（saved user ID，通常没有简写形式）。进程在运行过程中可以从有效UID切换到实际UID和已保存UID。（实际上Linux还有另外一个UID，即文件系统UID，但很少用到，代表访问文件系统的用户。）

典型的Setuid程序行为

实际UID可能会和我们以往的理解有冲突。我们也许会有疑问，为什么要经常和不同的用户

ID打交道？举个例子，假如你使用sudo启动了一个进程，如果要终止它，仍然需要用sudo，但这时候，你却无法使用你普通用户的身份来终止它。疑问产生了，这时你的普通用户不应该就是实际用户身份吗？为什么会没有权限终止程序呢？

产生这种情况，是因为sudo和很多其他setuid程序会使用setuid()这样的系统调用来显式地更改有效UID和实际UID。这样做是为了避免一些由于各UID不匹配导致的副作用和权限问题。

> **注解** 如果你对用户ID切换的细节和规则感兴趣，可以查看setuid(2)帮助手册，还可以参考SEE ALSO部分列出的其他帮助手册。这些帮助手册里涉及很多针对不同情况的系统调用。

有些程序不想把它们的实际UID设置成root。要防止sudo更改实际UID，可以将下面一行加入你的/etc/sudoers文件（请注意使用root运行程序可能带来的副作用）。

```
Defaults      stay_setuid
```

相关安全性

因为Linux内核通过setuid程序和相关系统调用来处理用户切换（以及相关的文件存取权限），系统管理员和开发人员必须特别注意以下两点。

- 有setuid权限的程序。
- 这些程序所执行的功能。

如果你创建了一个bash shell的副本，setuid为root，普通用户就可以运行它来获得整个系统的控制权，就是这么简单。另外，setuid为root的程序中的bug也有可能会为系统带来风险。攻击Linux系统的常见方式之一就是利用那些以root名义运行的程序的漏洞，这样的例子数不胜数。

由于攻击系统的手段众多，因此防止系统受到攻击也是一项十分繁重的任务。其中最有效的方式之一是强制使用用户名和密码进行验证。

7.9 用户标识和认证

多用户系统必须支持用户标识（identification）和用户认证（authentication），以保证基本的用户安全。*用户标识*判定用户的身份，即询问你是哪一位用户。*用户认证*让用户来证明自己就是声称的那位用户。此外，*用户授权*（authorization）用来限定用户的权限。

对于用户标识，Linux内核通过用户ID来管理进程和文件的权限。对于用户认证，Linux内核控制如何执行setuid，以及如何让用户ID执行setuid()系统调用来切换用户。然而，内核对于运行在用户空间中的有关用户认证的相关事宜却一无所知，比如用户名和用户密码等等。

我们在7.3.1节中介绍过用户ID和密码的对应关系，现在我们来介绍用户进程如何使用这些对应关系。我们先看一个简化的例子，即用户进程需要知道它的用户名（与有效UID对应的用户名）。在传统的Unix系统中，进程会通过以下步骤获得用户名。

(1) 进程使用geteuid()系统调用从内核处获得它的有效UID。

(2) 进程打开并浏览/etc/passwd文件。

(3) 进程从/etc/passwd文件中读取其中的一行。如果没有可读取的内容，则进程获取用户名失败。

(4) 进程将整行内容解析为字段（就是用冒号分隔的列）。第3列即是当前行的用户ID。

(5) 进程将第4步中获得的用户ID和第1步中的用户ID进行匹配，如果匹配成功，则该行的第1列即为要找的用户名，整个过程结束。

(6) 进程继续读取/etc/passwd文件中的下一行，并返回第3步。

事实上，这个过程比较长，实际执行起来要复杂得多。

为用户信息使用库

如果上述过程要让每个有此需求的开发人员自己实现的话，整个系统会变得支离破碎，错误百出，难以维护。万幸的是，一旦你从`geteuid()`获得用户ID，你需要做的就只是调用像`getpwuid()`这样的标准库函数来获得用户名。（这些调用的详细使用方法可参考帮助手册。）

有了共享的标准库，你可以自己对需要的功能做一系列的加工而不会影响到其他程序。比如，你可以不用/etc/passwd，而是使用LDAP这样的网络服务来获得用户名。

上述方法对于通过用户ID得到对应的用户名来说行得通，但是对于密码来说就不行了。我们在7.3.1节中介绍过，一般来说，经过加密的密码是/etc/passwd的一部分，如果你要验证用户输入的密码，你需要将用户输入的密码加密，然后和/etc/passwd文件中的密码进行比对。

这样的传统实现方式有下面几个局限。

- 加密协议并没有一个系统层面的统一标准。
- 它假定的前提是你需要对加密密码有访问权限。
- 它假定的前提是每当用户需要访问资源的时候，你都要让用户输入用户名和密码（这会让人抓狂）。
- 它假定的前提是你使用密码。如果你使用的是一次性标记、智能卡、生物识别技术或者其他形式的验证，你需要自己加入对它们的支持。

上述的一些局限也促成了影子密码机制的开发，我们在7.3.3节中介绍过。它就是建立系统层面的密码配置标准所迈出的第一步。不过上述大部分的问题促成了PAM解决方案的出现。

7.10　PAM

为了提高用户验证的灵活性，Sun Microsystems公司在1995年提出了一个新的标准，叫作可插入验证模块（Pluggable Authentiation Module，以下简称PAM）。它是一个共享的验证库（由Open Source Software Foundation RFC 86.0组织于1995年10月提出）。进行用户验证的时候，用户被提交给PAM来决定该用户是否能够成功完成验证。这样比较容易加入新的验证方式和技术，比如两段式验证和物理钥匙。除了对验证机制的支持，PAM还提供一些有限的验证控制服务（如可以对某些用户禁止cron这样的服务）。

因为用户验证的应用场景很多，所以PAM使用了一系列可以动态加载的验证模块。每个模块

负责一个具体的任务，比如pam_unix.so模块负责检查用户密码。

这样的任务很不简单。编程接口都很复杂，PAM看起来也没有能够解决所有的问题。无论怎样，Linux系统中涉及用户验证的程序基本上都是使用PAM，大部分Linux系统也是使用PAM。因为PAM是基于Unix现有的验证API，所以在集成PAM支持的时候只需少量的额外工作即可。

7.10.1　PAM配置

我们将通过PAM的配置来了解PAM的工作原理。PAM的应用配置文件通常存放在/etc/pam.d目录（在较老的系统中有可能是/etc/pam.conf文件）。目录中的文件很多，可能会让人找不到头绪。一些文件名应该会包含一些你熟知的系统名称，比如cron和passwd。

由于这些配置文件在不同的Linux系统上各异，我们很难找到一个通用的例子。我们以chsh的配置文件中的一行为例：

```
auth        requisite   pam_shells.so
```


该行表示用户的shell必须在/etc/shells中，以便使用户能够与chsh顺利进行验证。配置文件中每一行有三个字段，按顺序依次是功能类型、控制参数和模块。以下是它们代表的意思。

- 功能类型：是指某个用户应用程序请求PAM执行的任务。本例中是auth，即用户验证的任务。
- 控制参数：指定PAM在成功执行任务或者任务执行失败后的操作（本例中为requisite）。这一点我们稍后会详细介绍。
- 模块：指定该行所运行的验证模块。本例中pam_shells.so模块检查用户shell是否在/etc/shells中。关于PAM配置的详细内容，你可以查看pam.conf(5)帮助手册。现在先来看一些PAM的基础内容。

功能类型

用户应用程序可以请求PAM执行以下四类功能。

- auth：用户验证（验证用户身份）。
- account：检查用户账号状态（例如用户是否对某一操作有权限）。
- session：仅在用户当前进程内执行（例如显示当日的消息）。
- password：更改用户密码或其他验证信息。

对于任何配置行来说，功能类型和模块会共同决定PAM执行的操作。一个模块可以有多个功能类型，所以当我们查看配置行时，需要结合功能类型和模块来确定该行的功能。比如，pam_unix.so模块在执行auth时检查密码，但是在执行password时却是设置密码。

控制参数和堆栈规则

PAM的一个重要特性是它的配置行中使用堆栈定义的规则。也就是说，你可以为执行的操作定义一些规则。这也凸显了控制参数的重要性：某一行任务执行的成败会影响到后面的行甚至整个任务执行的成败。

控制参数有两类：简单语法和高级语法。简单语法的控制参数主要有以下三种。

- ❑ sufficient：如果该规则执行成功，用户验证即成功，PAM会忽略其他规则。如果规则执行失败，则PAM继续执行其他规则。
- ❑ requisite：如果该规则执行成功，PAM继续执行其他规则。如果规则执行失败，用户验证即失败，PAM会忽略其他规则。
- ❑ required：如果该规则执行成功，PAM继续执行其他规则。如果规则执行失败，PAM继续其他规则，但是无论其他规则执行结果如何，最终的验证将失败。

让我们继续上例，下面是chsh验证的一个堆栈实例：

```
auth        sufficient    pam_rootok.so
auth        requisite     pam_shells.so
auth        sufficient    pam_unix.so
auth        required      pam_deny.so
```

当chsh请求PAM执行用户验证时，根据以上配置，PAM执行以下步骤（见图7-4）。

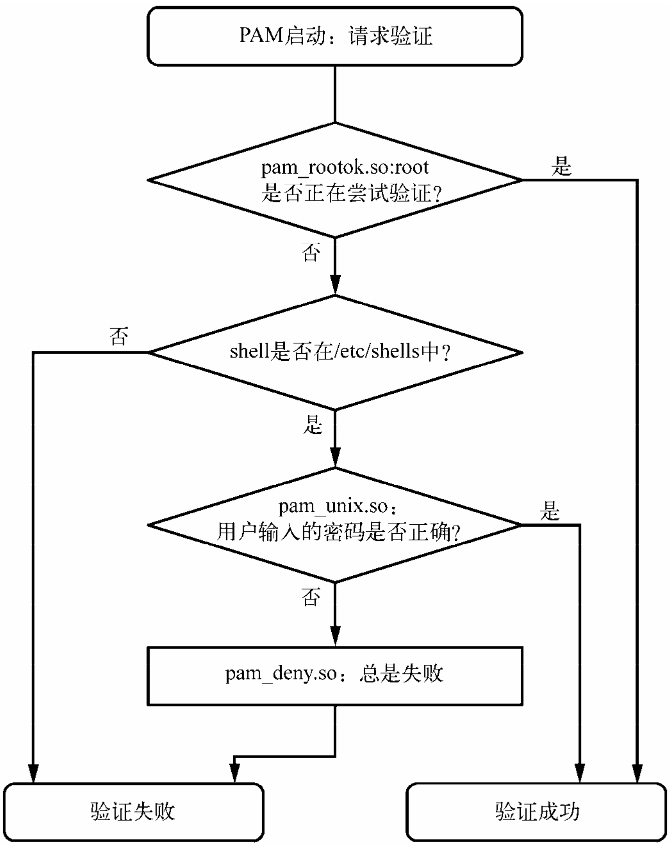


图7-4　PAM规则执行流程

(1) pam_rootok.so模块检查进行验证的用户是否是root。如果是的话，则验证立即通过并且忽略其他后续验证。这是因为控制参数是sufficient，表示当前操作执行成功，PAM会立即通知chsh验证成功。否则继续执行步骤2。

(2) pam_shell.so模块检查用户的shell是否在/etc/shells中。如果不是的话，模块返回失败，requisite控制参数表示PAM应立即通知chsh验证失败，并忽略其他后续验证。如果shell在/etc/shells中，模块返回验证成功，并且根据控制参数required继续执行步骤3。

(3) pam_unix.so模块要求用户输入密码并检查。控制参数为sufficient，意思是该模块验证通过后，PAM即向chsh报告验证成功。如果输入密码不正确，PAM继续步骤4。

(4) pam_deny.so模块总是返回失败，且由于控制参数为required，PAM向chsh返回验证失败。在没有其他规则的情况下，这是默认的操作。（请注意，required控制参数并不导致PAM立即失败，它还会继续后续的操作，但是最终还是返回验证失败。）

注解 *在讨论PAM的时候，不要将**功能**和**操作**混淆起来。功能是高层次的目标，即用户请求PAM执行的操作（诸如用户验证）。操作是PAM为了达到目标执行的某个具体任务。你只需要记住，用户应用程序首先执行功能，然后PAM负责执行相关操作。*

高级语法的控制参数使用方括号（[]）表示，让你能够根据模块的返回值（不仅仅是成功和失败两种）手动定义相应的操作。详情可以查看pam.conf(5)帮助文档。了解了简单语法控制参数后，高级语法控制参数就不是问题了。

模块参数

PAM模块能够在模块名后带参数。在pam_unix.so模块中你经常会看到类似下面的内容：

```
auth        sufficient   pam_unix.so   nullok
```

参数nullok表示用户可以不需要密码（如果用户没有密码则默认为验证失败）。

7.10.2 关于PAM的一些注解

由于其拥有控制流能力和模块参数语法，PAM配置语法具备了编程语言的某些特征和功能。目前我们仅仅是介绍了皮毛，下面是更多关于PAM的介绍。

- 可以使用man -k pam_(请注意此处的下划线)来列出系统中的PAM模块。要查看这些模块的存放位置可能会很难，你可以用locate unix_pam.so命令试试运气。
- 帮助手册中有每个模块相关功能和参数的详细信息。
- 很多Linux系统会自动生成一些PAM配置文件，所以尽量不要在/etc/pam.d目录中直接对它们进行更改。在做更改之前请阅读/etc/pam.d文件中的注释信息，如果它们是由系统生成的文件，注释中会对来源加以说明。
- /etc/pam.d/other配置文件中包含默认配置，用于那些没有自己的配置文件的应用程序。默认配置往往是拒绝所有的验证。

- 在PAM配置文件中包含其他配置文件的方法有很多种。可以使用`@include`语法加载整个配置文件，也可以使用控制参数来加载某个特定功能的配置文件。不同的系统有不同的加载方式。
- PAM配置不以模块参数结束。一些模块可以访问/etc/security中的其他文件，通常用来配置针对某个用户的特定限制。

7.10.3　PAM和密码

Linux密码校验近年来不断发展，留下了很多密码配置包，有些时候容易产生混淆。首先是/etc/login.defs这个文件，它是最初影子密码的配置文件。其中包含了用于影子密码文件加密算法的信息，不过使用PAM的新版本系统很少使用它了，因为PAM配置中自己包含了这些信息。但即便如此，有时候/etc/login.defs中的加密算法必须和PAM配置中的相匹配，以防万一有的应用程序不支持PAM。

PAM是从哪里获得密码加密信息呢？我们前面讲过，PAM处理密码的方式有两种：通过`auth`功能（用来校验密码）和`password`功能（用来设置密码）。查看密码设置参数很容易，最佳方式是使用`grep`，如下所示：

```
$ grep password.*unix /etc/pam.d/*
```

匹配的行中应该包含pam_unix.so，像下面这样：

```
password        sufficient      pam_unix.so  obscure sha512
```

参数`obscure`和`sha512`告诉PAM在设置密码时执行什么操作。首先，PAM检查密码是否足够复杂（也就是说，新密码不能和老密码太相似等等），然后PAM使用SHA512算法来加密新密码。

但这只在用户设置密码时生效，而不是在PAM校验密码时。PAM是怎样知道在用户验证时使用哪个算法呢？配置信息中没有这方面的信息，对于pam_unix.so的`auth`功能来说，没有针对加密信息的参数，帮助手册里也没有。

事实上，（直至本书成书时）pam_unix.so仅仅是靠猜测，通常是通过libcrypt库来逐一尝试每个文件，直至找出使用的那个算法，或者直至没有可尝试的文件。所以你通常不用太担心密码校验加密算法。

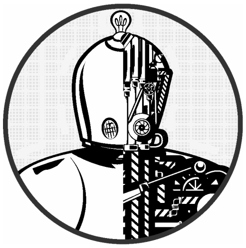
7.11　前瞻

至此我们已经到达的全书的一半，介绍了Linux系统的很多关键组成部分。特别是关于日志和用户的内容，让你了解到Linux系统是如何将服务和任务分解为小而独立的部分，并且仍然能够进行一定深度的交互。

本章主要是用户空间方面的内容，我们需要对用户空间进程和它们消耗的资源有一个新的认识。因此，在第8章中我们会讲解一些关于内核的深层次的内容。

第 8 章

进程与资源利用详解



本章我们将深入介绍进程之间的关系、内核和系统资源。计算机硬件资源主要有三种：CPU、内存和I/O。进程为获得这些资源相互竞争，内核则负责公平地分配资源。内核本身也是一种软件资源，进程通过它来创建新的进程，并和其他进程通信。

本章介绍的工具当中，有很多涉及性能监控。在你试图找出系统变慢的原因时，这些工具会非常有帮助。然而我们不需要过多关注系统性能，因为对已经运行良好的系统进行优化往往是浪费时间。我们应该更多地了解这些工具的功能，同时在此过程中我们会对内核有更深入的了解。

8.1　进程跟踪

我们在2.16节介绍过如何使用ps命令查看系统中运行的进程。ps命令列出当前运行的进程，但是无法提供进程随时间变化的情况，因而你无法得知哪个进程使用了过多的CPU时间和内存。

在这方面，top命令比ps命令更有用些，因为它能够显示系统的当前状态，还有ps命令结果显示的许多字段，并且每秒更新一次信息。但最重要的是，top命令将系统中最活跃的进程（即当前消耗CPU时间最多的那些进程）显示在最上方。

你可以通过键盘向top发送命令。下面是一些比较重要的键盘命令。

空格键	立即更新显示内容。
M	按照当前内存使用量排序。
T	按照CPU累计使用量排序。
P	按照当前CPU使用量（默认）排序。
u	仅显示某位用户的进程。
f	选择不同的统计信息来显示。
?	为所有top命令显示使用情况统计。

Linux中另外还有两个类似于top的工具，提供更详细的信息和更丰富的功能，它们是atop和htop。还有另外一些工具提供额外的功能，比如htop命令包含一些lsof命令的功能，其中lsof我们将在下节介绍。

8.2　使用 lsof 查看打开的文件

lsof命令列出打开的文件以及使用它们的进程。由于Unix系统大量使用文件，所以lsof在系统排错方面是最有用的命令之一。但lsof不仅仅显示常规文件，还显示网络资源、动态库以及管道等等。

8.2.1　lsof输出

lsof的输出结果通常信息量很大，如下面的例子中，你看到的只是其中一些片段。这一输出结果中包含init进程中打开的文件和运行中的vi进程：

```
$ lsof
COMMAND PID   USER   FD TYPE DEVICE  SIZE    NODE NAME
init      1   root  cwd DIR     8,1  4096       2 /
init      1   root  rtd DIR     8,1  4096       2 /
init      1   root  mem REG      8, 47040 9705817 /lib/i386-linux-gnu/libnss_files-2.15.so
init      1   root  mem REG     8,1 42652 9705821 /lib/i386-linux-gnu/libnss_nis-2.15.so
init      1   root  mem REG     8,1 92016 9705833 /lib/i386-linux-gnu/libnsl-2.15.so
--snip--
vi    22728 juser  cwd DIR     8,1 4096 14945078 /home/juser/w/c
vi    22728 juser   4u REG     8,1 1288 1056519  /home/juser/w/c/f
--snip--
```

其中包含以下字段。

- COMMAND：拥有文件描述符的的进程对应的命令名。
- PIM：PID。
- USER：运行进程的用户。
- FD：该列包含两种元素。本例中FD列显示文件的作用。该列还能够显示打开文件的描述符。文件描述符是一个数字，进程通过它使用系统库和内核来进行文件标识和操作。
- TYPE：文件类型（如常规文件、目录、套接字等）。
- DEVICE：包含该文件的设备的主要代码和次要代码。
- SIZE：文件大小。
- NODE：文件的索引节点编号。
- NAME：文件名。

以上各字段所有可能的值可以在帮助手册lsof(1)查看，不过只看输出结果应该就很清楚了。例如，FD列中使用加粗字体标出cwd的那些行，它们显示当前进程的工作目录。另一个例子是最后一行，它显示用户正在使用vi编辑的文件。

8.2.2 lsof的使用

运行lsof有以下两种基本方式。

- 输出完整的结果，然后将输出结果通过管道用命令less显示，然后在其中搜索你想要的内容。由于输出结果信息量很大，该方法可能会花点时间。
- 使用命令行选项来过滤lsof的输出结果。

你可以使用命令行选项提供一个文件名作为参数，然后使用lsof显示只和参数匹配的条目。例如，下面的命令显示/usr目录中所有打开的文件：

```
$ lsof /usr
```

根据PID列出打开文件，使用以下命令：

```
$ lsof -p pid
```

你可以运行lsof -h查看lsof所有的选项。大部分选项和输出格式有关。（请参考第10章中关于lsof网络特性的介绍。）

> **注解** lsof和内核信息密切相关。如果你升级内核时没有按常规升级系统的其他部分，你可能需要升级lsof。如果你同时升级了内核和lsof，新的lsof可能需要你重新启动新内核以后才能够正常工作。

8.3 跟踪程序执行和系统调用

到目前为止，我们介绍的工具都是针对运行中的进程。然而，有时候你的程序可能在系统启动后马上就终止了，这时你可能会一头雾水，lsof命令也不管用了。实际上，使用lsof查看执行失败的进程非常困难。

strace（系统调用跟踪）和ltrace（系统库跟踪）命令能够帮助你了解程序试图执行哪些操作。它们的输出信息量都很大，不过一旦你确定了查找的范围，有很多工具可以帮助你定位需要的信息。

8.3.1 strace命令

之前介绍过，系统调用是用户空间请求内存执行的经过授权的操作，诸如打开文件、读取数据等等。strace能够显示进程涉及的所有系统调用。可以通过下面的命令查看：

```
$ strace cat /dev/null
```

在第1章中我们介绍过，如果一个进程想启动另一个进程，该进程会使用fork()系统调用来

从自身创建出一个副本，然后副本调用exec()系统调用集来启动和运行新的程序。strace命令在fork()系统调用之后开始监控新创建的进程（即源进程的副本）。因而该命令输出结果的一开始几行应该显示execve()的执行情况，随后是内存初始化系统调用brk()，如下所示：

```
execve("/bin/cat", ["cat", "/dev/null"], [/* 58 vars */]) = 0
brk(0)                                  = 0x9b65000
```

输出的后续部分涉及共享库的加载。除非你想知道加载共享库的细节，否则你可以忽略这些信息。

```
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb77b5000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
--snip--
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200^\1"..., 1024)= 1024
```

除此之外，你可以跳过输出结果中的mmap，直到下面的部分：

```
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 6), ...}) = 0
open("/dev/null", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 3), ...}) = 0
fadvise64_64(3, 0, 0, POSIX_FADV_SEQUENTIAL)= 0
read(3,"", 32768)                       = 0
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
```

这部分内容显示正在运行中的命令。首先我们来看看open()调用，它用来打开文件。返回值3代表执行成功（3是内核成功打开文件后返回的文件描述符）。在它下面，你可以看到cat从/dev/null（read()中也包含文件描述符3）读取数据。在读取完所有数据后，程序关闭文件描述符，并调用exit_group()退出。

如果过程中发生错误会出现什么情况？你可以使用strace cat not_a_file命令来检查open()的执行情况：

```
open("not_a_file", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
```

上面显示open()无法打开文件，它返回-1代表出错。你可以看到，strace显示确切的错误代码，并对错误进行简短的描述。

Unix上的程序经常会遇到无法找到文件的情况。当系统日志和其他日志无法提供有帮助的信息，你也别无他法时，可以使用strace。strace甚至还可以用于那些已经和源进程分离的守护进程，如：

```
$ strace -o crummyd_strace -ff crummyd
```

上面的-o选项为所有由crummyd产生的子进程记录日志，并保存到crummyd.strace.pid文件中。其中pid是子进程的PID。

8.3.2 ltrace命令

ltrace命令跟踪对共享库的调用。它的输出结果和strace类似，所以我们在这里提一下。但是它不跟踪内核级的内容。请记住，共享库调用比系统调用数量多得多。所以你有必要过滤ltrace命令的输出结果。ltrace命令有很多选项可以帮到你。

> **注解** 有关共享库的细节可参考15.1.4节。ltrace命令对静态连接二进制库无效。

8.4 线程

在Linux中，一些进程被细分为更小的部分，我们称为线程（thread）。线程和进程很类似，它有一个标识符（即TID）。内核运行线程的方式和运行进程基本相同。但有一点不同，即进程之间不共享内存和I/O这样的系统资源，而同一个进程中的所有线程则共享该进程占用的系统资源和一些内存。

8

8.4.1 单线程进程和多线程进程

很多进程只有一个线程，叫单线程进程。有超过一个线程的叫多线程进程。所有进程最开始都是单线程，起始线程通常称为*主线程*。主线程随后可能会启动新的线程，这样进程就变为多线程。这个过程和进程使用fork()创建新进程类似。

> **注解** 对于单线程的进程来我们很少提及线程。本书中除非是遇到某些特定的多线程进程，否则我们不提及线程一说。

多线程的主要优势在于，当进程要做的事情很多时，多个线程可以同时在多个处理器上运行，这样可以加快进程的运行速度。虽然你也可以同时在多个处理器上运行多个进程，但线程相对进程来说启动更快，并且线程间通过共享的进程内存来相互通信，比进程间通过网络和管道相互通信更加便捷高效。

一些应用程序使用线程来解决在管理多个I/O资源时遇到的问题。传统上来说，进程有时候会使用fork()创建新的子进程来处理新的输入输出流。线程提供相似的机制，但却省去了启动新进程的麻烦。

8.4.2　查看线程

默认情况下，ps和top命令只显示进程的信息。如果想查看线程的信息，可以添加一个m选项，如下例所示。

例8-1　使用ps m查看线程

```
$ ps m
  PID TTY      STAT   TIME COMMAND
 3587 pts/3    -      0:00 bash❶
    - -        Ss     0:00 -
 3592 pts/4    -      0:00 bash❷
    - -        Ss     0:00 -
12287 pts/8    -      0:54 /usr/bin/python /usr/bin/gm-notify❸
    - -        SLl    0:48 -
    - -        SLl    0:00 -
    - -        SLl    0:06 -
    - -        SLl    0:00 -
```

该例显示进程和线程的信息。每一行有一个PID字段（其中❶、❷、❸行的PID值是数字），代表一个进程，和一般的ps输出一样。PID字段值为-的那些行代表进程中的线程。本例中进程❶和❷只有一个线程，进程❸（PID值为12287）有四个线程。

如果想要使用ps查看TID，需要使用自定义的输出格式。下面的例子显示PID、TID以及相关命令。

例8-2　使用ps m查看PID和TID

```
$ ps m -o pid,tid,command
  PID   TID   COMMAND
 3587     -   bash
    -  3587   -
 3592     -   bash
    -  3592   -
12287     -   /usr/bin/python /usr/bin/gm-notify
    - 12287   -
    - 12288   -
    - 12289   -
    - 12295   -
```

例8-2中显示的线程和例8-1中的相对应。请注意，单线程进程中的TID和PID相同，即主线程。对于多线程进程12287，线程12287是主线程。

> **注解**　和进程不同，通常你不会和线程进行交互。要和多线程应用中的线程打交道，你需要对该应用的具体实现非常了解，即使这样，我们也不推荐这种方式。

就资源监控而言，线程可能会带来一些麻烦，因为在多线程进程中，多个线程可能同时占用

资源。例如，top默认情况下不显示线程，你需要按H键来显示线程。我们马上将要介绍很多资源监控工具，它们需要一些额外的步骤来打开线程显示功能。

8.5 资源监控简介

现在我们来介绍一下资源监控，包括CPU时间、内存和磁盘I/O。我们将从系统和进程两个层面来了解。

很多人为了提高性能去深入了解Linux内核。然而，大部分Linux系统在默认配置下性能都不错，很有可能你花了很多时间来优化系统，但实际效果甚微。特别是在你对系统没有足够了解的时候更是如此。所以，与其使用各种工具来尝试性能优化，不如来看看内核如何在进程之间分配资源。

8.6 测量 CPU 时间

如果要监控进程，可以使用top命令加-p选项，如下所示：

```
$ top -p pid1 [-p pid2 ...]
```


使用time命令可以查看命令整个执行过程中占用的CPU时间。大部分shell提供的time命令只显示一些基本信息，所以你可能需要运行/usr/bin/time。例如，如果要查看ls命令占用的CPU时间，可以运行以下命令：

```
$ /usr/bin/time ls
```

time在ls结束后会显示像下面这样的结果，关键的字段我们使用粗体字标出：

```
0.05user 0.09system 0:00.44elapsed 31%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (125major+51minor)pagefaults 0swaps
```

其中三个关键字段分别代表以下含义。

- user：用户时间，指CPU用来运行程序代码的时间，以秒为单位。在现在的处理器中，命令的运行速度很快，有些不超过一秒，time命令会将它们四舍五入为0。
- system：系统时间，指内核用来执行进程任务的时间（例如读取文件和目录）。
- elapsed：消耗时间，指进程从开始到结束所用的全部时间，包括CPU执行其他任务的时间。这个数字在检测性能方面不是很有帮助，不过将消耗时间减去用户时间和系统时间所剩余的时间，能够让你得知进程等待系统资源所消耗的时间。

输出结果中的其余部分是有关内存和I/O使用的内容。在8.9节中，我们会讲解关于页面错误输出的内容。

8.7 调整进程优先级

你可以更改内核对进程的调度方式，从而增加或减少安排给进程的CPU时间。内核按照每个进程的调度优先级来运行进程，这些优先级用–20和20之间的数字表示。有些古怪的是，–20是最高的优先级。

ps -l命令显示当前进程的优先级，不过使用top命令更容易一点，如下所示。

```
$ top
Tasks: 244 total, 2 running, 242 sleeping, 0 stopped, 0 zombie
Cpu(s): 31.7%us, 2.8%sy, 0.0%ni, 65.4%id, 0.2%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 6137216k total, 5583560k used, 553656k free, 72008k buffers
Swap: 4135932k total, 694192k used, 3441740k free, 767640k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+ COMMAND
28883 bri       20   0 1280m 763m  32m S   58 12.7 213:00.65 chromium-browse
 1175 root      20   0  210m  43m  28m R   44  0.7  14292:35 Xorg
 4022 bri       20   0  413m 201m  28m S   29  3.4   3640:13 chromium-browse
 4029 bri       20   0  378m 206m  19m S    2  3.5  32:50.86 chromium-browse
 3971 bri       20   0  881m 359m  32m S    2  6.0 563:06.88 chromium-browse
 5378 bri       20   0  152m  10m 7064 S    1  0.2  24:30.21 compiz
 3821 bri       20   0  312m  37m  14m S    0  0.6  29:25.57 soffice.bin
 4117 bri       20   0  321m 105m  18m S    0  1.8  34:55.01 chromium-browse
 4138 bri       20   0  331m  99m  21m S    0  1.7 121:44.19 chromium-browse
 4274 bri       20   0  232m  60m  13m S    0  1.0  37:33.78 chromium-browse
 4267 bri       20   0 1102m 844m  11m S    0 14.1  29:59.27 chromium-browse
 2327 bri       20   0  301m  43m  16m S    0  0.7 109:55.65 unity-2d-shell
```

上面的输出结果中，PR（意思是优先级）字段显示内核当前赋予进程的调度优先级。这个数字越大，内核调用该进程的几率越小。决定内核分配给进程CPU时间的不仅仅是优先级，并且优先级在进程执行过程中也会根据其消耗的CPU时间而频繁改变。

PR字段旁边是NI（意思是优先值）字段，显示有关内核进程调度的少许信息，如果你想干预内核的进程调度，这个信息会对你有用。内核使用NI值来决定进程下一次在什么时间运行。

NI默认值是0。例如，你在后台运行一个计算量很大的进程，且不希望它影响到前台的交互，想让它在其他进程空闲的时候再运行，就可以使用renice命令将NI设置为20（pid是你要设置的进程的ID）：

```
$ renice 20 pid
```

如果你是超级用户，你可以将NI设置为一个负数，但这并不是一个好方法，因为系统级进程也许无法获得足够的CPU时间。事实上，你可能根本就不需要自己设置NI值，因为Linux系统大多时候是单个用户在使用，所以没有太多竞争。（从前多个用户使用一个系统的时候，NI值就重要得多。）

8.8 平均负载

CPU的性能比较容易来衡量。平均负载（load average）是准备就绪待执行的进程的平均数。也就是某一时刻可以使用CPU的进程数的一个估计值。系统中大多数进程通常把时间花在等待输入上（如从键盘、鼠标、网络等），这意味着这些进程没有准备就绪可执行，所以并不计入平均负载。只有那些真正在运行的进程才计入平均负载。

8.8.1 uptime的使用

uptime命令显示三个平均负载值和内核已经运行的时长：

```
$ uptime
... up 91 days, ... load average: 0.08, 0.03, 0.01
```

以上三个粗体数字分别代表过去1分钟、5分钟和15分钟的平均负载值。如你所见，系统并不很繁忙：过去15分钟只有平均数目为0.01的进程在运行。也就是说，如果你只有一个处理器，它过去15分钟只运行了用户空间应用的1%。（一般来说，大部分桌面系统的平均负载为0，除非你在编译程序或者玩游戏。平均负载为0通常是一个好迹象，说明CPU不是很忙，系统很省电。）

8

> 注解　目前桌面系统的用户界面组件比以往占用更多的CPU。例如在Linux系统上，Web浏览器的Flash插件可能消耗很多资源，一些蹩脚的Flash应用动辄占用大量的CPU和内存。

如果平均负载值接近1，说明某个进程可能完全占用了CPU。这时可以使用top命令来查看，通常出现在列表最上方的就是那个进程。

现在很多系统有多个处理器核心（或CPU），它们使得进程能够同时运行。如果你有两个核心并且平均负载为1，这意味着只有其中一个处于活跃状态，如果平均负载为2，说明两个都处于忙状态。

8.8.2 高负载

平均负载值高并不一定表示系统出现了问题。系统如果有足够的内存和I/O资源可以运行许多进程。如果平均负载值高的同时系统响应速度还很快，就不需要太担心，这说明系统中有很多进程在共享CPU。进程间需要相互竞争CPU时间，如果它们放任其他进程长时间占有CPU，它们自身的运行时间就会大大增长。对于Web服务器来说，高平均负载值也是正常现象，因为进程启动和结束得很快，以至于平均负载检测机制无法获得有效的数据。

然而，如果平均负载值高并且系统响应速度很慢的话，可能意味着内存性能有问题。当系统出现内存不足的情况时，内核会开始执行一个机械性的反复动作，或者说在磁盘和内存间交换进

程数据。此时很多进程会处于执行准备就绪状态，但是可能没有足够内存。这种情况下，它们保持准备就绪状态的时间要比平时久一些。

下面让我们了解一些有关内存的详细内容。

8.9　内存

查看系统内存状态最简单的方法之一是使用free命令，或者查看/proc/meminfo文件来了解系统内存被作为缓存和缓冲区的实际使用情况。我们前面介绍过，内存不足可能会导致性能问题。如果没有足够内存用作缓存和缓冲区（被其他程序占用）的话，也许你需要考虑增加内存。不过，把所有的性能问题都轻易归咎于内存不足也是不可取的。

8.9.1　内存工作原理

我们在第1章介绍过，CPU通过MMU（内存管理单元）将进程使用的虚拟地址转换为实际的内存地址。内核帮助MMU把进程使用的内存划分为更小的区域，我们称为*页面*。内核负责维护一个数据结构，我们称为*页面表*，其中包含从虚拟页面地址到实际内存地址的映射关系。当进程访问内存时，MMU根据此表将进程使用的虚拟地址转换为实际的内存地址。

进程执行时并不需要立即加载它所有的内存页面。内核通常在进程需要的时候加载和分配内存页面，我们称为*按需内存分页*（on-demand paging或者demand paging）。要了解它的工作原理，让我们来看一看进程是如何启动和运行的。

(1) 内核将程序指令代码的开始部分加载到内存页面内。

(2) 内核可能还会为新进程分配一些内存页面供其运行使用。

(3) 进程执行过程中，可能代码中的下一个指令在已加载的内存页面中不存在。这时内核接管控制，加载需要的内存页面，然后让程序恢复运行。

(4) 同样地，如果进程需要使用更多的内存，内核接管控制，并且获得空闲的内存空间（或者腾出一些内存空间）分配给进程。

8.9.2　内存页面错误

如果内存页面在进程想要使用时没有准备就绪，进程会产生*内存页面错误*（page fault）。错误产生时，内核从进程接管CPU的控制权，然后使内存页面准备就绪。内存页面错误有两种：轻微错误和严重错误。

轻微内存页面错误

进程需要的内存页面在主内存中但是MMU无法找到时，会产生轻微内存页面错误。通常发生在进程需要更多内存时，或MMU没有足够内存空间来为进程存放所有页面时。这时内核会通知MMU，并且让进程继续执行。轻微内存页面错误不是很严重，在进程执行过程中可能会出现。通常你不需要对此太在意，除非是那些对性能和内存要求很高的应用。

严重内存页面错误

严重内存页面错误发生在进程需要的内存页面在主内存中不存在时，意味着内核需要从磁盘或者其他低速存储媒介中加载。太多此类错误会影响系统性能，因为内核必须做大量的工作来为进程加载内存页面，占用大量CPU时间，妨碍其他进程的运行。

有一些严重内存页面错误是不可避免的，例如你第一次运行某个程序并从磁盘加载代码时，很可能会发生这种情况。如果你陷入了内存不足的状况，且内核开始在内存和磁盘间交换页面，以为新页面腾出空间，这就是比较棘手的情况了。

查看内存页面错误

你可以使用ps、top或time命令为某个进程的内存页面错误查找原因。下面是一个例子，列举time命令提供的内存页面错误信息。（cal命令的输出可以忽略，我们将其重定向到/dev/null。）

```
$ /usr/bin/time cal > /dev/null
0.00user 0.00system 0:00.06elapsed 0%CPU (0avgtext+0avgdata 3328maxresident)k
648inputs+0outputs (2major+254minor)pagefaults 0swaps
```

从以上加粗的信息可以看出，程序运行过程中产生了两个严重内存页面错误和254个轻微内存页面错误。严重内存页面错误发生在当内核首次从磁盘加载一个程序的时候。如果再次运行该程序，你可能不会再碰到严重内存页面错误，因为内核可能已经将从磁盘加载的内存页面放入缓存了。

如果你想在进程运行过程中查看产生的内存页面错误，可以使用top或者ps命令。可以使用top命令加f选项设置显示的字段，u选项显示严重内存页面错误的数目。（结果会显示在一个新的列nFLT中，轻微内存页面错误不显示。）

使用ps命令时，你可以使用自定义输出格式来查看某个进程产生的内存页面错误。下面是进程20365的一个例子：

```
$ ps -o pid,min_flt,maj_flt 20365
  PID  MINFL  MAJFL
20365 834182     23
```

MINFL和MAJFL列显示轻微和严重内存页面错误数目。在此基础上，你还可以加入其他的进程选择参数，请参见帮助手册ps(1)。

查看内存页面错误能够帮助你定位出现问题的组件。如果你对整个系统的性能感兴趣，你需要一个对所有进程的CPU和内存性能进行监控汇总的工具。

8.10 使用 vmstat 监控 CPU 和内存性能

在众多系统性能监控工具中，vmstat命令是较为陈旧的一个，但也是运行开销最小的一个。你可以使用它来清晰地了解内核交换内存页面的频率、CPU的繁忙程度以及IO的使用情况。

通过查看vmstat的输出结果能够获得很多有用的信息。下面是vmstat 2命令的输出结果，统

计信息每2秒刷新一次：

```
$ vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r b    swpd   free   buff  cache   si    so   bi    bo   in   cs us sy id wa
 2 0  320416 3027696 198636 1072568    0     0    1     1    2    0 15 2 83  0
 2 0  320416 3027288 198636 1072564    0     0    0  1182  407  636  1 0 99  0
 1 0  320416 3026792 198640 1072572    0     0    0    58  281  537  1 0 99  0
 0 0  320416 3024932 198648 1074924    0     0    0   308  318  541  0 0 99  1
 0 0  320416 3024932 198648 1074968    0     0    0     0  208  416  0 0 99  0
 0 0  320416 3026800 198648 1072616    0     0    0     0  207  389  0 0 100  0
```

输出结果可以归为这几类：procs（进程）、memory（内存使用）、swap（内存页面交换）、io（磁盘使用）、system（内核切换到内核代码的次数）以及cpu（系统各组件使用CPU的时间）。

上例中的输出结果是系统负载不大的一种典型情况。通常我们会从第二行看起，因为第一行是系统整个运行时期的平均值。上例中，系统有320 416 KB内存被交换到磁盘（swpd），有大约3 025 000 KB（3 GB）空闲内存（free）。虽然有一部分交换空间正在被占用，但是si（换入，swap-in）和so（换出，swap-out）列仍显示内核没有在磁盘交换内存。buff列显示内核用于磁盘缓冲区的内存（可参考4.2.5节）。

在最右边的cpu列，你可以看到CPU时间被分为us、sy、id和wa列。它们依次代表用户任务、系统（内核）任务、空闲时间和I/O等待时间占用的CPU时间的百分比。上例中运行的用户进程不多（只占用了最多1%的CPU时间），而内核基本上是空闲的，CPU 99%的时间均为空闲。

现在让我们来看看一个庞大的程序启动后会发生什么情况（前两行显示的是程序开始运行前一刻的信息）：

例8-3　内存活动

```
procs -----------memory-------- ---swap-- -----io---- -system-- ---- cpu----
r b   swpd   free   buff  cache   si   so   bi     bo  in  cs   us  sy id wa
1 0 320412 2861252 198920 1106804    0    0     0      0 2477 4481 25  2 72  0❶
1 0 320412 2861748 198924 1105624    0    0     0     40 2206 3966 26  2 72  0
1 0 320412 2860508 199320 1106504    0    0   210     18 2201 3904 26  2 71  1
1 1 320412 2817860 199332 1146052    0    0 19912      0 2446 4223 26  3 63  8
2 2 320284 2791608 200612 1157752  202    0  4960    854 3371 5714 27  3 51 18❷
1 1 320252 2772076 201076 1166656   10    0  2142   1190 4188 7537 30  3 53 14
0 3 320244 2727632 202104 1175420   20    0  1890    216 4631 8706 36  4 46 14
```

例8-3中的❶显示，CPU在一段时间内开始出现一定的使用量，特别是针对用户进程。因为可用内存充足，在内核越来越多使用磁盘的时候，缓存和缓冲空间的使用量开始增大。

随后我们看到一个有趣的现象：❷显示内核将一些被交换出（si列）磁盘的内存页面加载到内存中。这表示刚刚运行的程序有可能使用了一些和其他进程共享的内存页面。这种情况很常见，很多进程只有在启动时会使用到一些共享库代码。

请注意在b列中有一些进程在等待内存页面时被阻断（即被阻止运行）。简单说就是可用内存在减少，但是还未耗尽。上例中还有一些磁盘相关的进程，从bi和bo列不断增大的数字可以看出。

在内存耗尽的时候，输出结果则大为不同。内存耗尽时，因为内核需要为用户进程分配内存，缓冲区和缓存空间开始减少。一旦内存耗尽，内核开始将内存页面交换到磁盘，在so列中会开始出现进程，此时其他列的信息会相应变更。系统时间值会变大，数据交换更频繁，更多的进程处于阻断状态，因为它们所需的内存不可用（已经被交换出磁盘）。

我们还未介绍完vmstat的所有输出列。你可以查看vmstat(8)帮助手册以获得更详细的文档。不过建议你先从*Operating System Concepts*, 9th Edition（Wiley，2012）这类书或者其他渠道学习内核内存管理方面的知识。

8.11 I/O 监控

默认情况下，vmstat显示常用的I/O统计信息。虽然你可以使用vmstat -d获得十分详细的资源使用情况，但这一选项的输出信息十分庞大，会让人有些抓狂。我们可以从专门针对I/O的命令iostat看起。

8.11.1 使用iostat

和vmstat类似，iostat在不带任何参数时显示系统当前的运行时间信息：

```
$ iostat
[kernel information]
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           4.46    0.01    0.67    0.31    0.00   94.55

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda               4.67         7.28        49.86    9493727   65011716
sde               0.00         0.00         0.00    1230             0
```

上面avg-cpu部分和本章介绍的其他工具一样，显示的是CPU的使用信息，往下是各个设备的情况，如下所示。

- tps：平均每秒数据传输量。
- kB_read/s：平均每秒数据读取量。
- kB_wrtn/s：平均每秒数据写入量。
- kB_read：数据读取总量。
- kB_wrtn：数据写入总量。

iostat和vmstat的另一个相似之处是你可以设定一个间隔选项，如iostat 2，这样可以每隔2秒更新一次信息。间隔参数可以和-d选项配合使用，意思是只显示和设备相关的信息（如iostat -d 2）。

默认情况下，iostat的输出结果不包含分区信息。如果要显示分区信息，可以使用-p ALL选项。因为典型的系统上都会有很多分区，所以输出的信息量也会很大。下面是部分输出示例：

```
$ iostat -p ALL
--snip
--Device:      tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
--snip-
sda           4.67         7.27        49.83    9496139   65051472
sda1          4.38         7.16        49.51    9352969   64635440
sda2          0.00         0.00         0.00          6          0
sda5          0.01         0.11         0.32     141884     416032
scd0          0.00         0.00         0.00          0          0
--snip--
sde           0.00         0.00         0.00       1230          0
```

上例中，sda1、sda2和sda5均为sda磁盘上的分区，因而读和写两列的信息可能会有重叠。然而各分区的总和并不一定等于磁盘的总容量。虽然对sda1的读取同时也是对sda的读取，但请注意，从sda直接读取数据也是可能的，比如读取分区表数据。

8.11.2　使用iotop查看进程的I/O使用和监控

如果想要更深入地了解各个进程对I/O资源的使用情况，可以使用iotop工具，使用方法和top一样。它会持续显示使用I/O最多的进程，最顶端是汇总数据，如以下代码所示。

```
# iotop
Total DISK READ:      4.76 K/s | Total DISK WRITE:     333.31 K/s
  TID  PRIO USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
  260 be/3 root       0.00 B/s  38.09 K/s   0.00 %  6.98 % [jbd2/sda1-8]
 2611 be/4 juser      4.76 K/s  10.32 K/s   0.00 %  0.21 % zeitgeist-daemon
 2636 be/4 juser      0.00 B/s  84.12 K/s   0.00 %  0.20 % zeitgeist-fts
 1329 be/4 juser      0.00 B/s  65.87 K/s   0.00 %  0.03 % soffice.b~ash-pipe=6
 6845 be/4 juser      0.00 B/s  812.63 B/s  0.00 %  0.00 % chromium-browser
19069 be/4 juser      0.00 B/s  812.63 B/s  0.00 %  0.00 % rhythmbox
```

请注意，除了user、command、read、write列之外，还有一列叫TID（也就是线程ID），而非PID（即进程ID）。iotop是为数不多的显示线程而非进程的工具。

PRIO（意思是优先级）列表示I/O的优先级。它类似于我们介绍过的CPU优先级，但它还会决定内核为进程调度I/O读写操作分配多长时间。如果优先级为be/4，则be代表调度等级，数字代表优先级别。和CPU优先级一样，数字越小，优先级越高。比如内核会为be/3的进程分配比be/4的进程更多的时间。

内核使用调度等级为I/O调度施加更多的控制。iotop中有以下三种调度等级。

- be：即best-effort（尽力）。内核尽最大努力为其公平地调度I/O。大部分进程是在这类I/O调度等级下运行。
- rt：即real-time（实时）。内核优先调度实时I/O。
- idle：空闲。内核只在没有其他I/O工作的时候安排此类I/O工作。该调度等级不具备优先级。

你可以使用ionice工具来查看和更改进程的I/O优先级，也可以参考ionice(1)帮助手册。但一般情况下你不用关心I/O优先级。

8.12　使用 pidstat 监控进程

我们已经介绍过如何使用top和iotop来监控进程。但是，它们都会不断刷新输出结果，旧的输出会被新的覆盖。pidstat工具能够让你使用vmstat的方式来查看进程在过去某个时间段内的资源使用情况。下例是对进程1329监控结果，按秒记录。

```
$ pidstat -p 1329 1
Linux 3.2.0-44-generic-pae (duplex)     07/01/2015       _i686_ (4 CPU)

09:26:55 PM       PID    %usr %system  %guest    %CPU    CPU  Command
09:27:03 PM      1329    8.00    0.00    0.00    8.00      1  myprocess
09:27:04 PM      1329    0.00    0.00    0.00    0.00      3  myprocess
09:27:05 PM      1329    3.00    0.00    0.00    3.00      1  myprocess
09:27:06 PM      1329    8.00    0.00    0.00    8.00      3  myprocess
09:27:07 PM      1329    2.00    0.00    0.00    2.00      3  myprocess
09:27:08 PM      1329    6.00    0.00    0.00    6.00      2  myprocess
```

该命令在默认情况下显示用户时间和系统时间的百分比，以及综合的CPU时间百分比，还显示进程在哪一个CPU上运行。（%guest列有一点奇怪，指进程在虚拟机上运行时间的百分比。除非你是在运行虚拟机，否则可以忽略它。）

虽然pidstat默认显示CPU的使用情况，它还有其他很多功能。例如我们可以使用-r选项来监控内存，使用-d选项来监控磁盘。你可以自己尝试运行一下，更多针对线程、上下文切换和其他本章所涉及内容的选项可以参考pidstat(1)帮助手册。

8.13　更深入的主题

针对资源监控的工具有很多，其中一个原因是资源的种类很多，不同资源的使用方式不同。本章我们介绍了进程、进程中的线程和内核是如何使用CPU、内存和I/O等系统资源的。

另外一个原因是系统的资源是有限的，考虑到性能，系统中的各个组件都需要尽可能地减少资源消耗。过去很多用户共享一台计算机，所以需要保证每个用户公平地获得资源。现在的桌面系统都是单人使用，但是其中的进程仍然相互竞争以获取资源。高性能的网络服务器对系统资源监控的要求更高。

有关资源监控和性能分析的更加深入的主题有以下几方面。

- sar（即系统活动报告，System Activity Reporter）：sar包含很多vmstat的持续监控功能，另外还记录系统资源随时间使用情况。sar让你能够查看过去某一时刻的系统状态，这在你需要查看已发生的系统事件时非常有用。
- acct（即进程统计）：acct能够记录进程以及它们的资源使用情况。
- Quotas（即配额）：你可以将某些系统资源限制给某个进程或用户使用。可以到/etc/security/limits.conf中查看一些CPU和内存选项，帮助手册中也有limits.conf(5)文档。这是PAM的一个特性，只适用于那些通过PAM启动的进程（如登录 shell）。你还可以使用

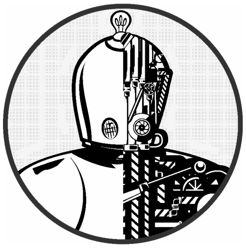
`quota`来限制用户可以使用的磁盘空间。

如果你对系统调度尤其是系统性能感兴趣，可以参考Brendan Gregg所著*Systems Performance: Enterprise and the Cloud*（Prentice Hall，2013）一书。

关于网络监控和资源使用，我们还有很多工具未介绍。在使用这些工具之前，你需要先了解网络的工作原理，我们将在下一章介绍。

第 9 章

网络与配置



网络能实现计算机之间的连接与数据收发。听起来够简单的，但想理解个中运作原理，你要先弄懂下面两个基础问题。

- 发送方怎么知道要发往哪里？
- 接收方怎么知道接收了什么？

而答案就是：计算机会用一系列负责接收、发送和识别数据的组件来完成这个过程。这些组件被划分在不同的组，而这些组层叠起来才形成一套完整的系统，叫作*网络层次*。Linux内核处理网络通信的方式类似第3章的SCSI子系统。

因为各层倾向于相互独立，所以这些组件的组合方案是多种多样的。这就是网络配置的难点所在。因此，本章先考察一种简单的分层。你将学到如何查看自己的网络设置，而当你明白每层的工作时，你就可以学习如何为自己的层次做配置。最后，你会接触一些更高级的内容 ，比如建立自己的网络以及配置防火墙。（如果对该部分无兴趣，也可以先跳过，需要时再回头看。）

9.1 网络基础

在讲解理论之前，先看下图9-1中的简单网络。

这种网络很普遍，大多数家庭和小型办公室都是这样配置的。该网络中的每台机器都叫作*主机*，它们都能与*路由器*相连。路由器也是一种主机，它使网络与网络之间能传输数据。这些机器（主机A、B、C）和路由器组成了一个*局域网*（Local Area Network，以下简称LAN）。LAN中的连接可以是有线的也可以是无线的。

图中的路由器还连接了一个云状物——互联网。因此LAN上的机器能通过路由器连接到互联网。本章的目标之一就是解释路由器如何实现这种连接。

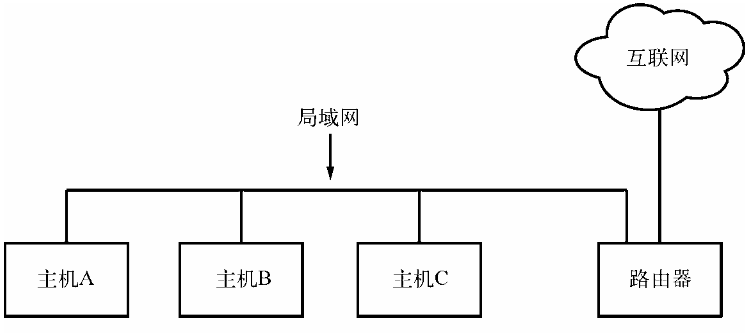


图9-1 一个通过路由器连接到互联网的典型局域网

下面先从LAN中安装了Linux的机器开始，如上图中的主机A。

数据包

数据被分成一块块的在网络上传输，每一块叫作一个*数据包*（packet）。包中分为两部分：*头*和*净荷*。头含有一些识别信息，如发送方、接收方以及基本的协议。净荷则含有实际需要传送的数据，如HTML或图片数据。

分包使主机能够交替地做出发送、接收、处理数据的动作，所以主机之间看似能同时通信。而且这样也令错误检测和重传数据更方便。

大多数情况下，你没必要担心数据包的解析，因为操作系统带有这样的功能。然而，理解数据包在以下介绍的网络层次中扮演什么角色还是有用的。

9.2 网络层次

一个功能完整的网络，包含一整套被称为*网络栈*的网络层次。典型的互联网栈，从顶部到底部如下所示。

- *应用层*：包含应用间、服务器间的交流语言——通常是一种高级的协议。一般有超文本传输协议（Hypertext Transfer Protocol，以下简称HTTP，用于Web）、安全套接层（Secure Socket Layer，以下简称SSL）、文件传输协议（File Transfer Protocol，以下简称FTP）。协议能够组合使用。例如SSL就常跟HTTP一起用。
- *传输层*：用于规定应用层的数据传输形式。该层包括数据完整性的检查、端口功能以及将数据分包（如果应用层未分包）。传输控制协议（Transmission Control Protocol，以下简称TCP）和用户数据报协议（User Datagram Protocol，以下简称UDP）是传输层最常见的协议。这层有时也被称为*协议层*。
- *网络层或网际层*：规定如何识别源主机和目的主机。IP（网际协议）规定了互联网所使用的包传输规则。因为本书只讨论互联网，所以我们只谈及IP。然而，网络分层就是为了做

到与硬件无关，所以，你可以在同一台机器上配置不同的网际层（如IP、IPv6、IPX、AppleTalk等）

- 物理层：规定如何通过物理中介（如以太网、调制调解器）发送原始数据。这层有时也被称为链路层或网络接口层。

理解网络栈的结构是很重要的。因为你的数据在到达目的地之前，至少要穿越它两次。举个例子，若你要在图9-1中从主机A发数据到主机B，你的字节会经历主机A的应用层、传输层、网际层，下至物理层，经过中介，然后再由下至上穿越主机B中的这些层次。如果需要通过路由器发送到互联网上的其他主机，那途中还会经历路由器或其他设备中的网络层次（这里的层次通常少些）。

有时一些层次会以特别的方式来干涉其他层的数据，因为按顺序地经历各层次不一定是高效的做法。例如，为了高效地过滤和转发数据，一些以往只负责物理层的设备，现在也会去检查传输层和网际层的数据。（现在只讲基础，你还不用担心这些问题。）

下面看看Linux机器如何连接到网络，以解答本章开头“发送方怎么知道要发往哪里”的问题。这里涉及栈底的物理层和网际层。然后再看看上面的两层，以解答“接收方怎么知道接收了什么”的问题。

注解　也许你听说过开放系统互连（OSI）参考模型，它是一种七层的网络模型，常用于教学和设计网络。但我们这里只提到直接会用到的四层。想知道更多分层方面的东西，可看Andrew S. Tanenbaum和David J. Wetherall所著的*Computer Networks*, 5th edition（Prentice Hall，2010）。

9

9.3　网际层

物理层太底层了，难理解，不如从网际层讲起，这比较好理解。我们现在所用的互联网，仍然基于网际协议第四版（IPv4），尚未彻底推行IPv6。网际层的重点之一就是，它与硬件或操作系统无关，也就是说，你能使用任何一种硬件或操作系统在互联网上收发数据包。

互联网的拓扑结构是无中心的。它由更小的网络——子网构成。意思是，各子网以某些方式相连。如图9-1的LAN就是一个子网。

一个主机可以置于不止一个子网中。正如你在9.1节看到的，负责将数据从一个子网送到另一子网的那种主机叫作路由器（也叫网关）。

图9-2对图9-1进行了补充，将该LAN标识为一个子网，并给路由器和每个主机标上了网络地址。图中路由器有两个地址，一个用于本地子网的10.23.2.1和另一个用于互联网的“上行地址”（这个地址在这里并不重要）。我们先看下那些地址，以及子网的标识。

互联网中的每台主机至少各有一个形如a.b.c.d的数字IP地址，如10.23.2.37。这种记法叫作点分四组序列。如果有个主机连接了多个子网，它在每个子网中都至少有一个IP地址。每个主机的IP地址在互联网中应该是唯一的。但你即将看到，在专用网络和网络地址转换（NAT）的情况中，

它会有点混乱。

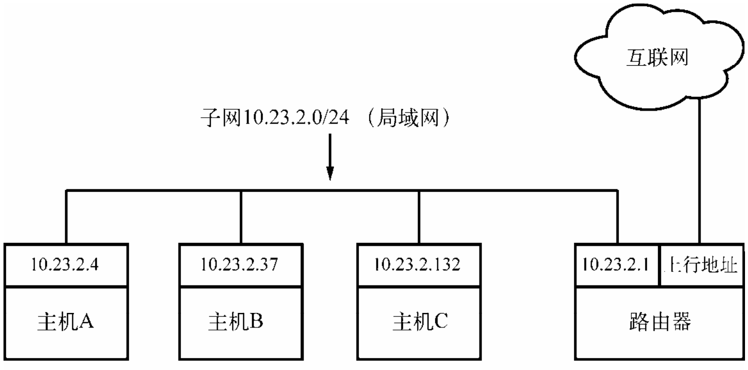


图9-2　带有IP地址的网络

注解　从技术上来说，一个IP地址由4个字节（32位）组成。现以abcd指代，a和d的范围都是1~254，b和c都是0~255。计算机按字节的原始形式处理IP地址。但人们为了方便，将其记为10.23.2.37这种格式，而不是0x0A170225。

IP地址在某种程度上就像邮编。想跟其他主机交流，你先要知道其IP地址。

让我们先来看看自己机器的地址。

9.3.1　查看自己计算机的IP地址

一个主机可以有多个IP地址，以下命令用于查看主机使用中的地址：

```
$ ifconfig
```

它会输出很多结果，其中应该包括以下内容：

```
eth0      Link encap:Ethernet  HWaddr 10:78:d2:eb:76:97
          inet addr:10.23.2.4  Bcast:10.23.2.255 Mask:255.255.255.0
          inet6 addr: fe80::1278:d2ff:feeb:7697/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:85076006 errors:0 dropped:0 overruns:0 frame:0
          TX packets:68347795 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:86427623613 (86.4 GB) TX bytes:23437688605 (23.4 GB)
          Interrupt:20 Memory:fe500000-fe520000
```

`ifconfig`能展现网际层和物理层的许多细节（有时不止一个网络地址），以后会对它们进行详解。现在关注一下第二行，它显示该主机有一个IPv4的地址10.23.2.4(`inet addr`)，还有掩码

（Mask）是 255.255.255.0。这是子网掩码，定义了IP地址的所属。下面看它是怎么回事。

注解 ifconfig之类（还有route、arp等）的命令，现已被ip取代。ip功能更强大，更适合写脚本时使用。可大部分人还是喜欢在交互式操作中使用旧的命令，而且还有其他版本的Unix兼容这些旧命令，所以我们还是用它们来讲解。

9.3.2 子网

子网就是一组相互连接的、带有按序排列的IP地址的主机。通常这组主机会在同一个物理网络中。如图9-2所示，10.23.2.1至10.23.2.254的主机可以构成一个子网，甚至10.23.1.1至10.23.255.254的都可以。

划分子网需要考虑两点：一个是*网络前缀*，一个是*子网掩码*（上一节中的ifconfig的Mask）。假若你要建立一个包含10.23.2.1到10.23.2.254的子网，那么它们*通用*的网络前缀就是10.23.2.0，子网掩码就是255.255.255.0。下面解释为什么是这样。

前缀和掩码是怎样表示子网中的可用IP的？将点分四组序列转换成位，会比较易懂。掩码标记了子网内主机能共用的IP位。现将10.23.2.0和255.255.255.0转换成位：

```
10.23.2.0:        00001010 00010111 00000010 00000000
255.255.255.0:    11111111 11111111 11111111 00000000
```

再将前缀中与掩码的那些1对应的位加粗：

10.23.2.0:	**00001010 00010111 00000010** 00000000

可将剩下未加粗部分的任何位设为1，以获得可用的IP地址（除了全0和全1）。

把它们合在一起，你就知道一个IP为10.23.2.1、子网掩码为255.255.255.0的主机是如何跟其他以10.23.2打头的主机共处一个子网的了。这整个子网可记为10.23.2.0/255.255.255.0。

9.3.3 共用子网掩码与无类域内路由选择

幸运的话，你只会接触到像255.255.255.0或255.255.0.0这种简单的子网掩码。不幸的话，你可能会遇到难以判断子网可用IP范围的那种，例如255.255.255.192。此外，你还可能会看到另一种子网的表示方式——*无类域内路由选择*（Classless Inter-Domain Routing，以下简称CIDR），它会将10.23.2.0/255.255.255.0写成10.23.2.0/24。

继续以上一节提到的二进制掩码为例来讲解CIDR。你会发现几乎所有子网掩码都是一堆1，后接一堆0。像255.255.255.0就是24个1后接8个0。掩码在CIDR中只记录其*开头*的1的个数，因此，10.23.2.0/24已完整地表达了前缀与掩码。

表9-1展示了一些子网掩码及其CIDR形式

表9-1　子网掩码

长形式	CIDR形式
255.0.0.0	8
255.255.0.0	16
255.240.0.0	12
255.255.255.0	24
255.255.255.192	26

注解　如果你不擅长进制转换，你可以用bc或dc这样的计算命令来辅助。例如在bc中输入obase=2;240，能输出240的二进制形式。

识别子网和其中的主机只是第一课，接下来你还要学会将子网连接起来。

9.4　路由和内核路由表

将各子网连通的过程，就是识别路由器的过程。回到图9-2，IP地址为10.23.2.4的主机A能够直接访问本地网络10.23.2.0/24上的其他主机。当要访问互联网上的其他主机时，它需要通过IP为10.23.2.1的路由器。

Linux内核是如何区分两个不同类型的目标地址的呢？其实，它是通过一种叫路由表的配置文件来决定自身的路由行为的。`route -n`命令可以显示出路由表。以下是10.23.2.4这种简单的主机可能拥有的路由表：

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags  Metric Ref    Use  Iface
0.0.0.0         10.23.2.1       0.0.0.0         UG     0      0      0    eth0
10.23.2.0       0.0.0.0         255.255.255.0   U      1      0      0    eth0
```

最后两行显示路由信息。Destination列是网络前缀，Genmask列是对应的掩码。这里有两个网络：0.0.0.0/0（所有IP都可在这里找到对应）和 10.23.2.0/24。每个网络的Flags列都有个U，意味着该路由是活动的。

这些Destination的差异在于Gateway与Flags的组合。0.0.0.0/0的Flags有G，表示需要通过Gateway（10.23.2.1）才能访问它。而10.23.2.0/24的Flags无G，表示它能被直接访问。该行的Gateway列记为0.0.0.0，其他列暂时不管。

还有个微妙的细节：如果想发信给10.23.2.132，而发现10.23.2.132又都在0.0.0.0/0和10.23.2.0/24之中，那么内核怎么判断应该用第二个呢？看Destination最大的那个就行。这里CIDR就帮大忙了：10.23.2.0/24符合，且它共用24个IP位；0.0.0.0/0也符合，但它没有共用的IP位（即0个）。所以，优先使用10.23.2.0/24。

注解 -n选项指定显示IP地址而非主机名和网络名。这个选项值得记住，因为其他网络命令如netstat也有该项。

默认网关

0.0.0.0/0能匹配互联网中的所有IP，它是默认路由，其Gateway列（route -n所示的）的地址是默认网关。如果目标地址没有匹配到其他网络，那就将信息从默认网关送出，到默认路由里找。你可以给主机取消默认网关的配置，那它将只能访问路由表中所剩的网域。

注解 掩码255.255.255.0的网络多数会给路由分配1这个地址（例如0.23.2.0/24中的10.23.2.1）。但这只是约定，当然也有例外。

9.5 基本 ICMP 和 DNS 工具

现在是时候来看一些操作主机的实用工具了。它们涉及两种协议：用于查找路由和连接问题的网际控制报文协议（Internet Control Message Protocol，以下简称ICMP），以及使用名称代替IP地址以便人们记忆的域名系统（Domain Name Service，以下简称DNS）。

9.5.1 ping

ping（见http://ftp.arl.mil/~mike/ping.html）是最常见的网络调试工具之一。它发送一个ICMP请求报文给一台主机，这会使该接收者回送报文。比如你运行ping 10.23.2.1，然后得到这样的输出信息：

```
$ ping 10.23.2.1
PING 10.23.2.1 (10.23.2.1) 56(84) bytes of data.
64 bytes from 10.23.2.1: icmp_req=1 ttl=64 time=1.76 ms
64 bytes from 10.23.2.1: icmp_req=2 ttl=64 time=2.35 ms
64 bytes from 10.23.2.1: icmp_req=4 ttl=64 time=1.69 ms
64 bytes from 10.23.2.1: icmp_req=5 ttl=64 time=1.61 ms
```

这里第一行说的是你发了56（包括头部的话，是84）字节的数据包给10.23.2.1（默认一秒一包），接下来的行是10.23.2.1的应答信息。其中最重要的是序列号（icmp_req）和来回时间（time）。返回信息的容量是发送信息的容量加8（数据包的内容不用管）。

漏掉的序列号（如2和4之间的3）通常说明连接有点问题。另外，如果回送的次序是乱的，那也是有问题的，因为现在是一秒一包，间隔算长的了，而如果有些包超过一秒才回收到，那说明连接很慢。

来回时间是指数据包从发出到收回所经历的时间。如果包无法到到目的地，那么最后一站的路由器会返回host unreachable（无法到达主机）的信息。

有线局域网应该做到没有丢包和快速响应（上面的例子来自无线局域网）。测试自己的子网和ISP时也应如此。

> **注解**　因为一些安全性问题，互联网上有些主机是不会响应ICMP请求的，所以你可能打得开一些网站，但使用ping却无法收到响应。

9.5.2　traceroute

学到本章后面的路由知识时，你可能会需要traceroute这个基于ICMP的工具。执行traceroute host能显示数据包到达目标主机所走过的路。（traceroute -n 主机名不查找主机名。）

它的优点之一，就是能告诉你每个路由之间的回程用时，如以下片段：

```
 4  206.220.243.106  1.163 ms  0.997 ms  1.182 ms
 5  4.24.203.65  1.312 ms  1.12 ms  1.463 ms
 6  64.159.1.225  1.421 ms  1.37 ms  1.347 ms
 7  64.159.1.38  55.642 ms  55.625 ms  55.663 ms
 8  209.247.10.230  55.89 ms  55.617 ms  55.964 ms
 9  209.244.14.226  55.851 ms  55.726 ms  55.832 ms
10  209.246.29.174  56.419 ms  56.44 ms  56.423 ms
```

因为6和7之间的有个很大的延时，所以那部分可能是一个长途的连接。

traceroute的输出信息可能会不连续，在某步出现超时，而后面的步又能正常显示。那很可能是那一步的路由拒绝返回traceroute想要的调试信息，而往后的路由却愿意返回结果。此外，一些路由或者会推迟处理那些调试请求，而优先处理那些不带调试的。

9.5.3　DNS与host

IP地址难记而且常更换，所以我们用www.example.com这种名字来指代它。你系统上的DNS库通常会帮你搞定这种转换，但有时你可能会想自己手动转换。host命令用于找出域名的IP地址：

```
$ host www.example.com
www.example.com has address 93.184.216.119
www.example.com has IPv6 address 2606:2800:220:6d:26bf:1447:1097:aa7
```

注意，这个例子同时显示了IPv4地址93.184.216.119和一个更长的IPv6地址，说明该主机已拥有一个下一代互联网地址。

你也可以用host来反查IP地址的域名，即输入IP地址而不是域名。不过这个不太可靠。因为不同的主机名可以指向同一个IP地址，所以DNS无法得知哪个主机名才是你要的。对此，域名管

理员必须手动建立反向的对应关系，但通常没人这么做。（DNS的问题远不是一个host命令能说清楚的，我们会在9.12节讲解基本的客户端配置。）

9.6　物理层与以太网

网际层的互联网只是一个*软件上*的概念。至今我们都还没讲到硬件方面的东西。虽说互联网的成功正是因为它无关硬件，能在任何计算机、任何操作系统、任何网络上通用，但不懂硬件的层面还是不行的。而这个硬件层面，就是物理层。

本书讨论最常见的一种物理层：以太网。在IEEE 802标准的家族中，以太网有很多种，从有线到无线，但它们总有一些共性，如下所列。

- 所有以太网上的设备都有各自的*介质访问控制*（Media Access Control，以下简称MAC）地址，有时这也叫*硬件地址*。它与主机的IP地址无关，而且它在主机所处的以太网中是唯一的（但在互联网这种大型的软件网络中不一定唯一）。MAC地址形如10:78:d2:eb:76:97。
- 以太网上的设备以*帧*的形式发送信息，帧里除了实际的数据，还有发送者和接收者的MAC地址。

以太网是一个单独的网络。假设你有一台主机连接了两个不同类型的以太网（两个不同的网络接口设备），你是无法直接使帧通过该主机从一个以太网传送到另一个以太网的，除非该主机建立了一种特殊的桥梁。而这就是网际层（如互联网层）的工作了。一般约定一个以太网就是一个互联网中的子网。如果要将帧送到另一个子网，工作在网际层的路由器会把它解包，重新封装，再发给目标，这也正是互联网的做法。

9

9.7　理解内核网络接口

网际层与物理层的连接方式必须使得网际层拥有不依赖于硬件环境的灵活性。Linux内核自有一套用于沟通这两层的方法，叫作（*内核*）*网络接口*。所谓配置网络接口，就是把网际层的IP地址跟物理层的硬件标识对应起来。网络接口的名字通常概括了它的硬件类型，例如eth0（计算机的第一块以太网卡）和wlan0（无线接口）。

在9.3.1节中，你已学到了查看和配置网络接口的命令ifconfig。回忆一下它的输出：

```
eth0     Link encap:Ethernet HWaddr 10:78:d2:eb:76:97
         inet addr:10.23.2.4 Bcast:10.23.2.255 Mask:255.255.255.0
         inet6 addr: fe80::1278:d2ff:feeb:7697/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
         RX packets:85076006 errors:0 dropped:0 overruns:0 frame:0
         TX packets:68347795 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:86427623613 (86.4 GB) TX bytes:23437688605 (23.4 GB)
         Interrupt:20 Memory:fe500000-fe520000
```

对于每一个网络接口，左边的是网络接口的名字，右边是有关它的一些配置和统计信息。这

里除了有我们之前讲过的网际层的信息，还显示了物理层的MAC地址（HWaddr）。包含UP和RUNNING的行表示该接口处于工作状态。

尽管ifconfig告诉了你关于硬件的信息（本例中你还看到了一些诸如中断和内存使用的底层信息），但它只被设计作查看和配置网路接口的初级工具。要想深入挖掘网络接口背后的硬件层次，展示或更改以太网卡的设定，我们可以用ethtool。（9.23节会简单介绍无线网络。）

9.8　配置网络接口

至此，网络栈底部的所有基本内容（物理层、网际层、Linux内核网络接口）已经涵盖。接着你还要做以下工作，把那些知识点串起来，帮助Linux机器连到互联网。

(1) 接上网络硬件，并保证内核有它的驱动。如果有，可以使用ifconfig -a来显示该硬件对应的网络接口。

(2) 进行任意物理层设置，如可以配置网络名称和密码。

(3) 给网络接口绑定一个IP地址和掩码，使得驱动（物理层）能识别子网（网际层）。

(4) 添加必要的路由，包括默认网关。

如果是一些连在一起的大型工作站，那会简单一些：第一步由内核来做，第二步不用做，第三步和第四步分别需要你用ifconfig和route命令来执行。

手动设置IP地址和掩码，需要执行以下命令：

```
# ifconfig interface address netmask mask
```

这里interface是网络接口的名字，例如eth0。当网络接口能用了的时候，就可以添加路由了（其实就是设置默认网关）：

```
# route add default gw gw-address
```

参数gw-address是你默认网关的IP地址，它必须处于你其中一个网络接口的地址和掩码所定义的子网之中。

手动添加和删除路由

以下命令用来删除默认网关：

```
# route del -net default
```

你可以在默认网关的位置填上其他路由。假如你机器在子网10.23.2.0/24，其中的路由器在10.23.2.44，你想访问192.168.45.0/24，那么以下命令让我们可以通过10.23.2.44访问 192.168.45.0：

```
# route add -net 192.168.45.0/24 gw 10.23.2.44
```

删除对某子网的路由时，则无需填写“路由”参数：

```
# route del -net 192.168.45.0/24
```

在深入了解路由之前，你就应该知道，它虽然表面上看起来很简单，但实际上很复杂。对于上例，你还必须确保192.168.45.0/24上的主机能看到10.23.2.0/24上的主机，否则你为192.168.45.0/24添加路由器也是没用的。

一般来说，你应该尽量让事情简单一点，对网络进行妥善设置，让主机只有一个默认路由。如果你有多个子网并想通过路由联通它们，最好是将这些路由器配置成各子网的默认网关。（你会在9.17节看到相关例子。）

9.9　开机启动的网络配置

我们已经讲过手动配置网络的各种方法，而验证网络配置可用的传统方法，就是在开机时让初始化程序执行一个含有`ifconfig`和`route`之类的网络配置命令的脚本，使它处于开机的一连串事件当中。很多服务器仍是采取这种做法。

我们曾经尝试过多种方法以使开机启动网络配置文件规范化。例如`ifup`和`ifdown`命令，（理论上）开机脚本可以运行`ifup eth0`，给eth0接口执行相应的`ifconfig`和`route`命令。不幸的是，不同的发行版对`ifup`和`ifdown`的实现各有不同，以至于配置文件也不一样。就像Ubuntu的ifupdown套装是用/etc/network的配置文件，但Fedora则用/etc/sysconfig/network-scripts。

你无需知道这些配置文件的细节。但如果你真的想手写配置而不用你发行版自带的配置工具，可以查查帮助手册，如ifup(8)和interfaces(5)。其实很少人这么做。它基本上只见于localhost的网络接口配置，因为对于现代的系统来说这样很不灵活。

9.10　手动和开机启动的网络配置带来的问题

以前大多数系统（现在还有很多）都在开机时配置网络，但现代的网络的动态特性意味着一台机器的IP地址不是一成不变的。与其将IP地址和其他网络信息存放在本机上，不如在接入网络时从其他地方获取。大部分网络客户端应用并不在乎你用什么IP，只要能通就行。动态主机配置协议（Dynamic Host Configuration Protocol，以下简称DHCP，9.16节有详细介绍）的工具可以给某些客户端做一些简单的网际层配置。

不过这还不是全部的。例如，无线网络还有更多的配置，诸如网络名称、权限、加密技术等。当你看得更全面一点，你会发现你的系统要应付如下问题。

- 对于带有多个网络接口的机器（如笔记本电脑的有线网卡和无线网卡），怎样选择用哪个网卡？
- 如何接入网络？对于无线网络，这包括扫获网络名称、选择网络名称、获取权限。
- 接入以后，怎样配置那些软件网络层，如互联网层？

- 如何让用户选择接入哪个网？比如说如何让用户接入一个无线网？
- 被断开时，机器应该做些什么？

普通的开机脚本做不到上面所说的，另外，手动做也很麻烦。正确的做法是使用系统服务来监控物理网络，根据一系列规则选择出用户需要的内核网络接口。该服务还需交互，使得用户无需使用极具破坏力的root账号也能在不同环境选择想要的网络。

9.11　一些网络配置管理器

基于Linux的系统都有一些自动配置网络的方法。桌面端或笔记本端常用NetworkManager。其他网络配置管理程序则针对更小型的嵌入式系统，例如OpenWRT的netifd、Android的ConnectivityManager服务、ConnMan和icd等。下面我们简单介绍下NetworkManager，因为它比较受欢迎。我们不会讲得太细，因为如果你有宏观的认识，那么你应该很容易理解NetworkManager和其他配置管理器。

9.11.1　NetworkManager的操作

NetworkManager是一个开机时系统就启动的守护进程。跟其他守护进程一样，它不依赖桌面组件。它的任务是监听系统和用户的事件，并根据一堆规则来改变网络配置。

NetworkManager运行的时候，维护着两个层次的配置。第一层是它从内核收集而来，并通过监控桌面总线（D-Bus）的udev来维护的一堆可用的硬件设备的信息。第二层是更具体的一些连接配置：硬件设备和额外的物理层、网际层的配置参数。例如，一个无线网络就可看作是一个连接。

NetworkManager激活一个连接的做法，是将任务交给其他特定的网络工具或守护进程（例如`dhclient`，它会从接上的本地网络那里获得网际层配置）。因为网络配置工具和方案因不同发行版而异，所以NetworkManager需要插件来适配它们，而非强推自己的标准。例如Debian/Ubuntu和Red Hat都有对应的插件。

NetworkManager启动时，会收集所有可用网络设备的信息，在连接列表当中决定激活某一个。以下是它“作出决定”的方法。

(1) 优先连接可用的有线网络，若没有，则连无线网络。

(2) 对于无线网络，优先选择以前连过的。

(3) 如果有多个都是以前连过的，则选择最近一次连接的那个。

连接建立以后，就由NetworkManager维护，直至其断开，或被更好的网络取代（例如用着无线的时候接上了有线），或用户强行更换。

9.11.2　与NetworkManager交互

多数人是通过桌面右上（或右下）角的一个指示连接状态（有线连接、无线连接或无连接）的图标来跟NetworkManager交互的。当你点击它时，它会弹出一些选项，例如让你选择无线网络，

以及断开当前网络。这一图标在不同的桌面环境中会有所不同。

除了用图标，你还可以在shell中使用其他工具来查询和操控NetworkManager。使用不带参数的nm-tool，可以快速地列出你当前的连接状态，里面有网络接口及它们的参数。这有些类似ifconfig，但没有ifconfig详细，尤其是关于无线网络方面。

nmcli命令可以操控NetworkManager，它也很常见。帮助手册nmcli(1)有其详细介绍。

最后，nm-online命令会告诉你网络能不能用。如果能用，则返回退出码0，其他情况，返回非0。（第11章会详细介绍在shell脚本中使用退出码。）

9.11.3 NetworkManager的配置

一般NetworkManager的配置文件是放在/etc/NetworkManager目录中，里面有各种不同的配置文件。通常我们是看NetworkManager.conf这个文件。它的格式类似XGD风格的.desktop文件和微软的.ini文件，将一些键值对参数写在不同的小节中。几乎所有的配置文件都有[main]小节，用以定义插件。以下就是在Ubuntu和Debian 中激活ifupdown插件的例子：

```
[main]
plugins=ifupdown,keyfile
```

其他发行版的插件还有ifcfg-rh（Red Hat类的发行版）和ifcfg-suse（SuSE）。上例中的keyfile插件用于支持NetworkManager原生配置文件。使用它，你就能在/etc/NetworkManager/system-connections中看到系统能认出的连接。

大多数情况下，你无需更改NetworkManager.conf，因为更具体的设置是在其他文件中。

1. 不受管理的接口

虽然NetworkManager可以帮你管理大部分网络接口，但有时你可能也想让NetworkManager忽略某些网络接口。例如，localhost的配置是从来都不需要改变的，而且人们也希望在开机时就把它配置好，因为可能有其他服务需要用到它。所以大多数的发行版都会让localhost免受NetworkManager管理。

你可以用插件来让NetworkManager忽略某个网络接口。如果你在用ifupdown（在Ubuntu和Debian上），那么，将接口的配置加到/etc/network/interfaces文件中，然后在NetworkManager.conf文件的ifupdown小节，把managed设为false：

```
[ifupdown]
managed=false
```

对于 Fedora或Red Hat的ifcfg-rh插件，则是在/etc/sysconfig/network-scripts的名为ifcfg-*的配置文件中找出这样的行：

```
NM_CONTROLLED=yes
```

9

如果没找到这样的行，或找到值为no的行，则代表NetworkManager会忽略该接口。通常ifcfg-lo文件就设好了是“忽略”的。你还可以指定忽略某个硬件地址：

```
HWADDR=10:78:d2:eb:76:97
```

如果这两类网络配置方案你都不使用，你还是可以通过keyfile插件，直接在NetworkManager.conf文件里指定要忽略的硬件地址，如下：

```
[keyfile]
unmanaged-devices=mac:10:78:d2:eb:76:97;mac:1c:65:9d:cc:ff:b9
```

2. 调度

NetworkManager配置的最后一个细节问题是，让其他系统事件与网络接口的开启或关闭产生关联。举个例子，有些关于网络的守护进程需要知道何时才开始或停止监听某个网络接口（如下一章讲到的安全shell守护进程）。

当网络接口的状态改变时，NetworkManager会运行/etc/NetworkManager/dispatcher.d里的所有东西，并给予一个up或者down的参数。这种是比较简单的，但很多系统都有自己的一套网络控制脚本，而不会将调度程序脚本放在那个目录里。就像Ubuntu，它有一个叫01ifupdown的脚本，用来运行/etc/network下某个子目录里的东西，如/etc/network/if-up.d。

这些脚本的细节，跟其余NetworkManager的配置一样，都是不重要的，重要的是记录下你修改过哪些配置，以便恢复。还有，要勤于查看系统中的脚本文件。

9.12　解析主机名

配置网络还有一个基础任务，就是用DNS解析主机名。 前文我们已讲过，用主机名解析工具host可以将主机名（如www.example.com）转换成IP地址（如10.23.2.132）。

DNS与之前的网络话题的不同之处在于它在应用层，完全是用户空间的事。技术上来说，它与本章所说到的物理层和网际层没什么关系。但若没有正确的DNS配置，你的连接也不会起作用。IP地址可能会改变，而且没人想记住一长串数字。自动的网络配置服务，如DHCP，几乎总是包含DNS的配置。

差不多所有Linux网络应用都会做DNS查找，步骤基本如下所示。

(1) 应用会调用一个函数，去查找主机名对应的IP。该函数在系统共享库中，应用只管调用，而无需了解其中实现手段。

(2) 当该函数运行时，它会根据一系列规则（规则在/etc/nsswitch.conf中）来决定查找的计划。例如，通常会有这样的规则——先检查/etc/hosts里的重写，再使用DNS。

(3) 使用DNS，先要找到DNS服务器（在一个额外的文件里定义了这个DNS服务器的IP）。

(4) 该函数发送一个DNS查询的请求给DNS服务器。

(5) DNS服务器将请求中主机名对应的IP返回给函数，再由函数返回给应用。

简单来说就是这样。典型的现代系统中还加入了其他东西来提高查找效率和增加扩展性，但现在我们先不涉及这些，只看看基本的内容。

9.12.1 /etc/hosts

大多数系统都允许你通过/etc/hosts重写主机名的查找，该文件看起来会是这样：

```
127.0.0.1    localhost
10.23.2.3    atlantic.aem7.net    atlantic
10.23.2.4    pacific.aem7.net     pacific
```

一般都会在这里看到localhost的条目（9.13节会介绍）。

注解 在以往的艰苦岁月里，人人都要复制一份集中的host文件到自己机器上，以便跟上IP的变更（详见RFCs 606 、608、623和625）。但随着 ARPANET和互联网的发展，已经不需要这样做了。

9.12.2 resolv.conf文件

传统的做法是在/etc/resolv.conf文件中指定DNS服务器。简单的话，会像以下这样，这里ISP的DNS服务器为10.32.45.23和10.3.2.3：

```
search mydomain.example.com example.com
nameserver 10.32.45.23
nameserver 10.3.2.3
```

search的那行定义了针对残缺主机名（仅有主机名的第一部分，如myserver而非myserver.example.com）的查找方式。这里指定了会到host.mydomain.example.com和host.example.com查找。但平常是没这么简单的。DNS配置现在有了很多改善。

9.12.3 缓存和零配置DNS

传统的DNS配置有两个主要的问题。第一个问题是，本机不缓存DNS服务器的回应，这样每次都请求解析会使网络减慢。为了解决这个问题，很多机器（包括作为DNS服务器的路由器）都会运行一个守护进程，尽可能地拦截DNS请求，并返回缓存中的DNS回应，做不到的话才把请求发送到DNS服务器。最常见的这种守护进程是dnsmasq和nscd。你还可以建立BIND（标准的Unix DNS守护进程）来做缓存。如果你在/etc/resolv.conf看到127.0.0.1（localhost），或者运行nslookup -debug host时看到127.0.0.1的话，那你就正运行着DNS缓存守护进程。

dnsmasq默认的配置文件是/etc/dnsmasq.conf，但你的发行版可能会重写它。在Ubuntu上，如果你用NetworkManager手动建立了一个连接，你会在/etc/NetworkManager/system-connections的某

个文件上找到它，因为当NetworkManager激活一个连接的时候，也同时按照那个配置来启动dnsmasq。（你可以通过去除NetworkManager.conf中对dnsmasq的注释来重写这个规则）。

第二个问题是扩展性差，查找域名还要应付一堆配置文件。假如你在你的网络中加了一套设备，你很可能想马上用域名来访问它。现在，零配置域名服务系统正是为解决这种问题而诞生，例如多播DNS（以下称mDNS）和简单服务发现协议（Simple Service Discovery Protocol，以下简称SSDP）。若你想在本地网络中按域名查找一个主机，那么可以发起广播，如果该主机存在，则它会告诉你它的IP。这些协议会在DNS解析之前工作，以告知有哪些服务可用。

Linux用得最多的mDNS实现是Avahi。你会经常看到/etc/nsswitch.conf把mdns作为解析器，而这个/etc/nsswitch.conf就是接下来要讲的。

9.12.4　/etc/nsswitch.conf文件

/etc/nsswitch.conf文件掌管着一些域名相关的优先级设定，例如账号和密码。但本章只谈及它的DNS设定。你系统中的该文件应该有这样的一行：

```
hosts:          files dns
```

files在dns之前，表明先查找/etc/hosts，后查找DNS服务器。通常这种做法是可行的（尤其是查找localhost，下文会提到），但你的/etc/hosts应尽可能简短，以免带来性能问题。小型专用网里的主机名当然可以放进去。（/etc/hosts还可在开机初期、网络不可用时帮助解析localhost。）

注解　DNS是个很大的话题，如果你的工作需要处理域名，可参考Cricket Liu和Paul Albitz所著*DNS and BIND*, 5th edition（O’Reilly，2006）。

9.13　Localhost

ifconfig的输出里还有个lo：

```
lo    Link encap:Local Loopback
      inet addr:127.0.0.1  Mask:255.0.0.0
      inet6 addr: ::1/128 Scope:Host
      UP LOOPBACK RUNNING  MTU:16436  Metric:1
```

lo接口是一个虚拟的网络接口，它叫作环回（loopback），因为它指向的是自己。连接127.0.0.1，其实就是连接本机。当发送数据到内核网络接口lo，内核只会将其重新包装，并通过lo回复。

开机脚本所用到的静态网络配置一般就是lo环回接口。例如，Ubuntu的ifup会读取/etc/network/interfaces中的lo，而Fedora会用/etc/sysconfig/network-interfaces/ifcfg-lo。你还可以用grep在/etc下的配置文件中挖掘到很多关于环回设备的配置。

9.14 传输层：TCP、UDP 和 Service

到目前为止，我们只看到了数据包如何在互联网的主机之间传递，即本章开头的“发送方怎么知道要发往哪里”。那么现在就该谈谈“接收方怎么知道接收了什么”了。计算机如何将所接收的数据包呈现给进程，这是个很重要的问题。内核擅长处理数据包，但用户空间程序则不擅长。另外，灵活性也是要考虑的：不同的应用要能同时地使用网络（例如你可以同时运行邮件应用和其他一些网络客户端）。

*传输层协议*能让网络层的包与应用层的需求无缝连接。其中最常用的两个协议就是TCP（传输控制协议）和UDP（用户数据报协议）。我们主要讲一下TCP，因为它是用得最多的，但同时也会简要地讲一下UDP。

9.14.1 TCP端口与连接

TCP方式是指不同的网络应用使用不同的网络*端口*。端口只是一个数字。如果把IP比作大厦的地址，那么端口就是大厦里不同的邮箱号码，是一种更细的划分。

使用TCP时，应用会在本机的一个端口和远程机器的一个端口之间建立*连接*（不要跟NetworkManager的连接混淆）。例如，网页浏览器会在本机的36406端口和远端的80端口间建立连接。从该应用的角度来看，36406是本地端口，80是远程端口。

一个连接可以用“IP加端口”代码来标识。要想查看你机器上的连接，可用netstat。这里展示了一些TCP连接：选项-n表示不用对主机名进行DNS解析，而-t则是只输出TCP连接。

```
$ netstat -nt
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address          Foreign Address         State
tcp        0      0 10.23.2.4:47626        10.194.79.125:5222      ESTABLISHED
tcp        0      0 10.23.2.4:41475        172.19.52.144:6667      ESTABLISHED
tcp        0      0 10.23.2.4:57132        192.168.231.135:22      ESTABLISHED
```

Local Address和Foreign Address列位的内容是针对你的机器而言的。也就是说，你的机器有个网络接口使用了IP地址10.23.2.4，其中端口7626、41475和57132都已连接了。第一行的意思是，47626与10.194.79.125的5222端口连接了。

9.14.2 建立TCP连接

要建立传输层的连接，进程会发送一系列特别的数据包，先初始化一个从其本机端口到远端端口的连接。为了识别这个连接请求并进行答复，远端必须要有个进程监听着那个被请求的端口。通常请求方被称为*客户端*，监听方被称为*服务器*（第10章有更详细的介绍）。

你应该了解，客户端挑选的本地端口是当前未用到的，而远端端口则是“公认的”。回忆上一节讲的netstat的输出：

```
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.23.2.4:47626         10.194.79.125:5222      ESTABLISHED
```

按刚才所说的规则，这是一个由本机发起的连接，因为本机端口看上去像是随便挑选的，而远端端口则是Jabber或XMPP消息服务的公认端口5222。

注解　动态挑选的端口叫作暂时端口。

而如果本机端口是公认的端口，那这个连接就可能是从远端发起的。下面例子中，172.24.54.234的远端连接着本机的80端口（默认Web端口）。

```
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.23.2.4:80            172.24.54.234:43035     ESTABLISHED
```

远端与你的公认端口连着，意味着你机器上有服务器正在监听着该端口。想对此进行确认，可用netstat列出你的机器正在监听的所有TCP端口。

```
$ netstat -ntl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:53            0.0.0.0:*               LISTEN
--snip--
```

local address有0.0.0.0:80的那行，说明本机正监听着来自所有远端对自己80端口的连接。（服务器可以限制某些接口的访问，正如最后一行，某个服务只监听着localhost对53的连接。）想知道更多，可用lsof来识别是哪个进程正在进行监听（会在10.5.1节介绍lsof）。

9.14.3　端口的数字和/etc/services

如何得知哪些端口是公认端口？这个问题没有标准答案，但有个不错的判定方法，是去看看/etc/services。这是一个纯文本文件，保存着一些常用端口及其服务的名字，如下所示。

```
ssh             22/tcp                  # SSH Remote Login Protocol
smtp            25/tcp
domain          53/udp
```

第一列是服务名称，第二列是端口及其特定的传输协议（可以不是TCP）。

注解　除了/etc/services，还有一个由RFC6335网络标准文档管理的端口在线注册网站http://www.ana.org/。

Linux中只有超级用户运行的进程才能使用1到1023的端口。其余用户可以监听或建立1024及以上端口的连接。

9.14.4　TCP的特点

作为一种传输层协议，TCP之所以流行是因为它对应用没什么要求。应用的进程只需要知道如何打开（或监听）、读取、写入和关闭连接即可。对应用来说，这一过程就像收发数据流一样，而进程的工作就跟处理文件一样简单。

然而，还是有很多"幕后"工作的。其一，TCP实现需要知道怎样将发出的数据流分装成数据包。比这麻烦的，是要知道怎样将收到的一系列包转换成进程所需的数据流，尤其是当这些包无序到来时，更为麻烦。其二，使用TCP的主机还要做错误检查：数据包在经由互联网传输的过程中可能会丢失或受损，TCP必须进行纠正。图9-3简要描绘了主机使用TCP的方式发送信息的过程。

幸好，你不需要很了解其中的细节，只要知道Linux的TCP实现基本是在内核当中，而且涉及内核数据结构。你的认识水平只要达到类似9.21节所讲的内容即可。

9.14.5　UDP

UDP比TCP简单得多。单条信息就可以构成一次传输，没有数据流的说法。跟TCP不同的是，UDP不纠正数据包的丢失或乱序。事实上，尽管UDP也有端口的概念，但不会用此来建立连接！一台主机从其某个端口发信息到一台服务器的某个端口，然后该服务器想回就回，不想回就不回。UDP确实有查错的功能，但并不要求接收方纠错。

若把TCP看成是打电话，那么UDP就像是寄信、发电报或发即时消息（当然即时消息稍微可靠些）。用UDP的应用，看重的是速度——尽快发出消息。它们不愿承担TCP的开销，因为它们已假设两台机之间的网络是可靠的。它们不需要TCP的纠错功能，因为它们有自己一套纠错系统，或他们根本不在乎出错。

使用UDP的一个例子就是网络时间协议（Network Time Protocol，以下简称NTP）。客户端发送一个简短的请求给服务器，以获取当前时间，而服务器的回复也同样简短。因为客户端希望尽可能快地得到回复，所以使用UDP是很合适的。如果服务器的回复传丢了，那么客户端可以再请求，或放弃。另一个例子是视频通话。它用UDP的方式发送图像，如果有些丢失了，接收方会尽力修补。

注解　本章剩余部分是一些进阶的网络话题，例如网络过滤、路由器等。它们属于较低的网络层级：物理层、网际层和传输层。如果你对这些内容不感兴趣，可直接跳到下一章：用户空间的应用层。你会从用户进程的角度看待网络的使用，而不仅仅是停留在地址和数据包的传输层面。

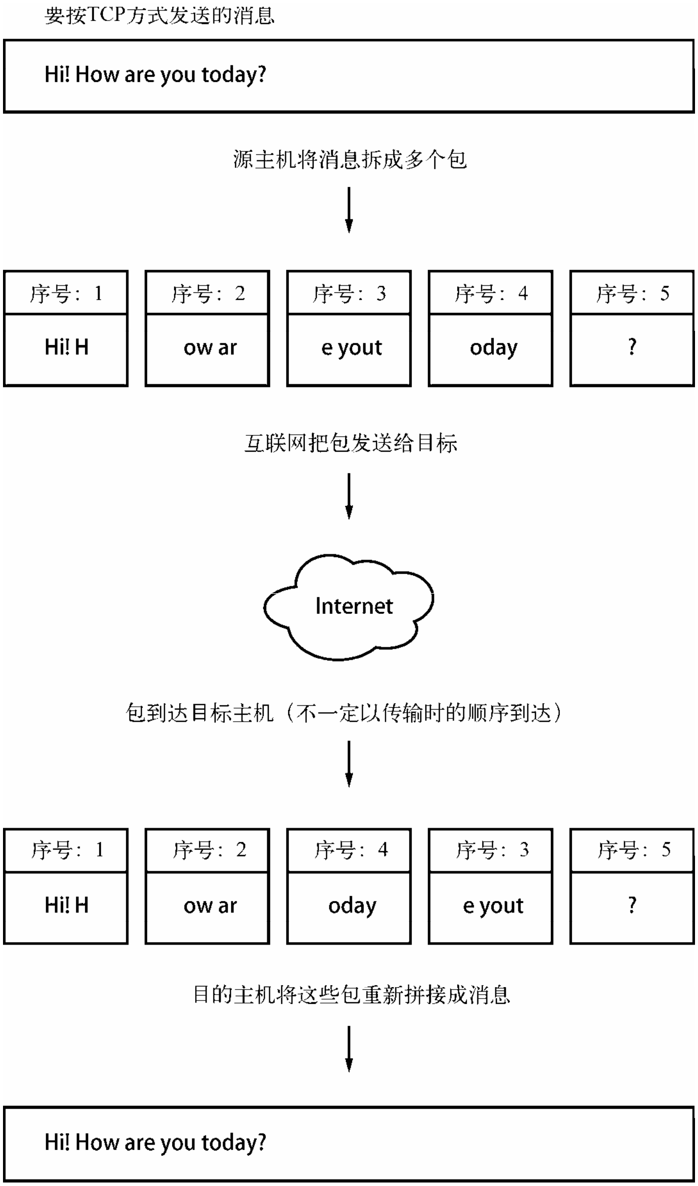


图9-3　用TCP的方式发信息

9.15　普通本地网络

下面来看看9.3节所讲到的普通网络的另外一些组件。回忆下，这个网络由作为子网的LAN和连接子网与互联网的路由器构成。而现在你要学的是以下内容：

- 子网中的主机如何自动获取网络配置；
- 如何建立路由；
- 路由器究竟是什么；
- 如何知道子网该用什么IP；
- 如何建立防火墙，以过滤来自互联网的无用信息。

我们先从“子网中的主机如何自动获取网络配置”开始。

9.16　理解 DHCP

所谓“自动获取网络配置”，就是让主机使用*动态主机配置协议*（Dynamic Host Configuration Protocol，以下简称DHCP）来获取IP、子网掩码、默认网关和DNS服务器。除了无需手动设置网络配置的好处以外，DHCP还可以帮助网络管理员防止IP冲突，以及减轻网络变更带来的影响。现在很少有不使用DHCP的网络。

主机必须先能发信至其网络中的DHCP服务器，才能通过DHCP获取网络配置。因此每个物理网络必须有自己的DHCP服务器，而在普通的网络（9.3节讲的那种）中，我们通常以路由器充当DHCP服务器。

注解　第一次发送DHCP请求时，主机是不知道DHCP服务器的地址的，所以它通常会把请求广播给网络中的所有主机。

主机只向DHCP服务器要求在一定时间内“租用”某个IP。当租用期结束，DHCP客户端可以要求更新租约。

9.16.1　Linux的DHCP客户端

尽管网络管理系统有很多种，但它们几乎全都使用互联网软件联盟（Internet Software Consortium，以下简称ISC）的`dhclient`程序来工作。你可以在命令行手动测试`dhclient`，但在此之前，你必须移除所有默认网关。这个测试需要指定网络接口的名字（这里以eth0为例）：

```
# dhclient eth0
```

开始后，`dhclient`会将它的进程ID放在/var/run/dhclient.pid中，而将租用信息放在/var/state/dhclient.leases中。

9.16.2　Linux的DHCP服务器

你可以在Linux机器上运行DHCP服务器，以妥善地管理IP地址。然而，除非你管理着一个拥有多个子网的大型网络，否则你最好还是选择内置DHCP服务器的路由器。

关于DHCP服务器，最重要的一点可能是：一个子网最好只有一个DHCP服务器，以防止IP冲突和配置错误。

9.17 将 Linux 配置成路由器

你可以把路由器看成是一种不止一个网络接口的计算机。将Linux机器配置成路由器不是什么难事。

假设你有两个LAN子网10.23.2.0/24和192.168.45.0/24，你可以用一个有三个网络接口（两个用于LAN、一个用于连接互联网的上行线）的Linux路由器连通它们。如图9-4，这跟之前讲过的简单网络的例子没太大区别。

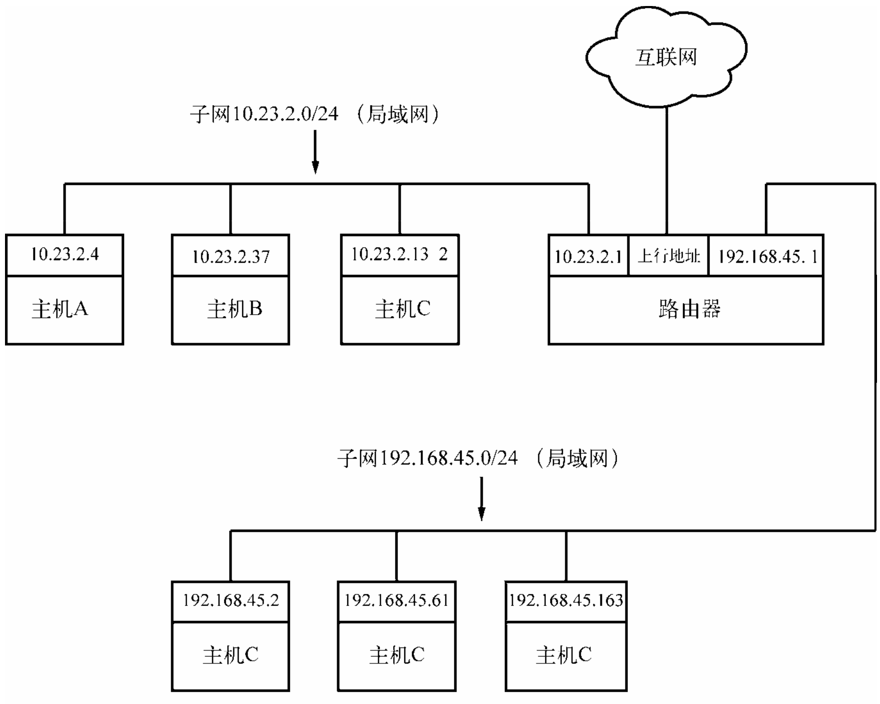


图9-4 以路由器连接的两个子网

该路由器在两个LAN子网的IP分别为10.23.2.1和192.168.45.1。配置好IP后，它的路由表大概会如下所示（现实中的接口名字可能会不一样；可以暂时忽略用于互联网的IP）：

```
Destination     Gateway      Genmask           Flags  Metric Ref     Use  Iface
10.23.2.0       0.0.0.0      255.255.255.0     U      0      0         0  eth0
192.168.45.0    0.0.0.0      255.255.255.0     U      0      0         0  eth1
```

现在两个子网上的主机都可将该路由器作为默认网关了（10.23.2.0/24的是10.23.2.1，而

192.168.45.0/24的是192.168.45.1）。如果10.23.2.4想发数据包给10.23.2.0/24以外的任何机器，它会将数据包传给10.23.2.1。例如，想从10.23.2.4（主机A）发数据包到192.168.45.61（主机E），数据包会经过路由器（10.23.2.1）的eth0进入路由器，再从eth1出去。

然而，默认情况下，Linux内核不会自动地在子网之间进行包传递。为了使基本路由功能生效，你需要用以下命令激活路由器内核的IP转发：

```
# sysctl -w net.ipv4.ip_forward
```

一旦你输入这个命令，而两个子网的主机又懂得将数据包发往该机器的话，那么该机器就能在两个子网间实现路由功能。要使它即使重启也能如旧，你可以在/etc/sysctl.conf文件里加入这个命令。有些发行版允许你把它写到/etc/sysctl.d的某个文件里，以使升级时不覆盖掉该命令。

互联网的上行线

当该路由器有第三个接口作为互联网的上行线，并也已设置好时，两个子网中的使用该路由作为默认网关的所有主机就都能连上互联网了。但这也使得事情变复杂了。因为像10.23.2.4这样的IP地址是处于所谓的“私有网络”中，对于互联网来说是不可见的。为了连上互联网，你必须在路由器上建立网络地址转换（Network Address Translation，以下简称NAT）的功能。几乎所有的路由器软件都支持这功能，所以这里也不例外。下面我们再详细看一下私有网络。

9

9.18　私有网络

假设你想建立你自己的网络，且你已准备好了机器、路由器和网络硬件，那么按你目前对一个普通网络的认识水平，你接下来应该会问：“我应该用什么IP子网？”

如果你想让你的IP在互联网上可见，你应该从ISP那里买一个。然而，因为IPv4的地址数非常有限，所以要买的话，花费会比较高，而让互联网上的其他人看到你服务器的IP也没什么用。大多数人是不需要这样做的，因为他们只作为客户端访问互联网。

以往，比较经济的替代方案是采用互联网标准文档RFC 1918/6761（如表9-2）定义的那些私有子网。

表9-2　RFC 1918和6761定义的私有网络

网　　络	子网掩码	CIDR形式
10.0.0.0	255.0.0.0	10.0.0.0/8
192.168.0.0	255.255.0.0	192.168.0.0/16
172.16.0.0	255.240.0.0	172.16.0.0/12

你可以随意对私有子网进行分割。除非你想让254以上的主机组成单个网络，不然采用本章举例的10.23.2.0/24那样的小子网就可以。（掩码为24的网络，也叫C类网络。尽管这种叫法有点过时，但依然是有意义的。）

这是什么意思呢？私有网络对互联网来说是不可见的，而互联网也不会直接发数据包到私有网络里。如果没有其他辅助的话，私有网络中的主机是无法跟外界沟通的。在互联网上拥有真实IP地址的路由器，需要一些方法来沟通私有网络和互联网。

9.19　网络地址转换（IP 伪装）

NAT是把单个IP分享给整个私有网络的常见方法，而且一般它在家庭和小型办公网络环境中都可用。Linux中，人们用得最多的NAT是IP伪装。

NAT背后的基本思想是，路由器做的不只是将数据包从一个子网传到另一个子网，而且还将数据包转化了。互联网上的主机只看到该路由器，而看不到其背后的私有网络。私有网络里的主机无需什么特别的配置，默认网关就使用该路由器。

这个系统大概是按下面的方式运作的。

(1) 私有网络里的一个主机想与外界通信，所以它会通过路由器发出连接请求的数据包。

(2) 路由器拦截了该请求，而不是传出去。（因为传出去的话是没用的，外界不能用它私有网络的IP找到它。）

(3) 路由器根据该数据包中关于目标的信息，开启其自身与该目标的连接。

(4) 连接成功后，路由器就伪造一个“连接已建立”的信息返回给内网主机。

(5) 现在路由器成了内网主机和远程主机之间的中介。远程主机并不知道该内网主机的存在，它只和路由器接头。

但实际上没这么简单。一般IP路由知道的就只有源和目标的IP地址。然而，如果路由器只有网际层，那么分处内网和外网的两台主机就不能在同一时间建立多个连接，因为网际层不能区分同一个源对同一个目标的不同的应用请求。因此，NAT必须在网际层之前就对数据包进行解剖，以得到更多标识信息（尤其是获取传输层的UDP和TCP端口号）。UDP比较简单，因为它只获取端口号，而不建立连接。但TCP就复杂了。

想将Linux机器当成NAT路由器，就要激活以下所有内核配置：网络包过滤（“防火墙支持”）、连接记录、IP表支持、全NAT以及MASQUERADE目标支持。大多数发行版都自带这些功能。

接着你还要运行一些看起来挺复杂的`iptables`命令，以使路由器为私有子网提供NAT功能。下面就是`eth1`接内网、`eth0`接外网的NAT例子（`iptables`的详细语法将在9.21节介绍）：

```
# sysctl -w net.ipv4.ip_forward
# iptables -P FORWARD DROP
# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
# iptables -A FORWARD -i eth0 -o eth1 -m state --state ESTABLISHED,RELATED -j ACCEPT
# iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
```

注解　尽管NAT在实际中应用广泛，但记住，它只是应付IPv4地址贫乏的一个技巧。在美好的将来，我们所有人都会用上IPv6（下一代互联网）的更大、更复杂的地址空间，而不用再作任何转换。

除非是软件开发者，否则一般不会用到以上命令，尤其是现在已经有了那么多功能各异的路由器产品。但Linux在网络中的角色不止这些。

9.20 路由器与 Linux

以往带宽小的时候，人们是使用各自的宽带连接互联网的。但后来，随着带宽增大，就发展成使用一台运行NAT的路由器来共用一条宽带连接出去。

制造商们为了响应新的市场需求，推出了制定的路由硬件，其中包含高效的处理器、闪存、多个网络口——足以给普通的网络提供DHCP、NAT等重要服务。在软件方面，则选用Linux。他们添加必要的内核功能，抽掉一些用户软件，再创建一个图形化的管理界面。

这类路由器一面世，就吸引了很多人去搞硬件。有个制造商，Linksys，被要求开放其某个软件的源代码，开放之后，便出现了一个Linux发行版，叫作OpenWRT。（WRT是Linksys的某个产品型号。）

除了出于兴趣，还是有其他原因促使人们采用这些发行版的：它们通常比制造商的固件要稳定，尤其是对于老旧的路由器，而且它们一般有功能加强。例如，想桥接一个无线网络的话，很多制造商都会要你买匹配的硬件，但有了OpenWRT，你就不再需要担心品牌的差异或者硬件的新旧。因为该操作系统是开放的，只要里面添加了对某硬件的支持，就可运行该硬件。

你可以使用本书的知识来测试那些制定的Linux固件，虽然它们各不相同，尤其是登录时的不同。因为他们很多都是嵌入式的，所以开源固件倾向于使用BusyBox来提供shell功能。BusyBox是一个单独的可执行程序，能让我们使用`ls`、`grep`、`cat`、`more`等Unix命令。它功能有限，但节省空间。还有，嵌入式系统的开机初始化程序都是很简约的。然而，这些限制通常都不是问题，因为制定的Linux固件大多包含一个网页化的管理界面，跟原厂的差不多。

9.21 防火墙

路由器尤其应该包含某种防火墙，来为你的网络抵御一些不速之客的请求。防火墙是一种软件和（或）硬件的配置，处于互联网和小型网络之间的路由器之中，以保证小型网络免于来自互联网的攻击。你也可以给每一台机器都配备防火墙，这样每台机器都可以从数据包的级别筛选进出的数据（从应用层级别的话，往往是使用服务器的程序进行访问控制）。为每台机器配备防火墙，这叫作IP过滤。

系统可以在这些情况下过滤数据包：

- 接收数据包时；
- 发送数据包时；
- 或将数据包转发到其他主机或网关时。

没有防火墙的话，系统只会按原来的做法来处理数据包。防火墙会在上述的几种场合设立检查点。这些检查点会根据以下标准来丢弃、拒绝或者接受数据包：

- 源或目标的IP或子网；
- 源或目标的端口（传输层的信息）；
- 防火墙的网络接口。

防火墙能让你接触到Linux内核中处理IP包的子系统。现在我们就来看一下。

9.21.1　Linux防火墙基础

Linux中，防火墙的规则是链型的。一个表就是一套链。数据包在Linux网络子系统的各部分移动时，内核就会对包应用某套规则。例如，从物理层接收到一个新的包之后，内核就会根据输入的数据激活对应的规则。

这些数据结构由内核来维护。整个系统叫作iptables，可以使用用户空间的`iptables`命令来创建和修改规则。

注解　现在有了一个叫nfttables的新系统，用于取代iptables。但本文讲述防火墙还是以iptables为主。

因为规则表有很多，表里面的规则也很多，所以使得数据包的流动变得比较复杂。然而，你一般都只会接触到一个叫作filter（过滤）的表，它控制基本的包流动。该表里面有三个基本的链：对应包输入的INPUT、对应包输出的OUTPUT和对应包转发的FORWARD。

图9-5和图9-6简要展示了对数据包应用这些规则的流程。之所以有两个图，是因为要区分输入（图 9-5）和输出（图9-6）。如你所见，由网络输入的数据包可能会走INPUT而不走FORWARD及OUTPUT，而输出的包则不会走INPUT或FORWARD。

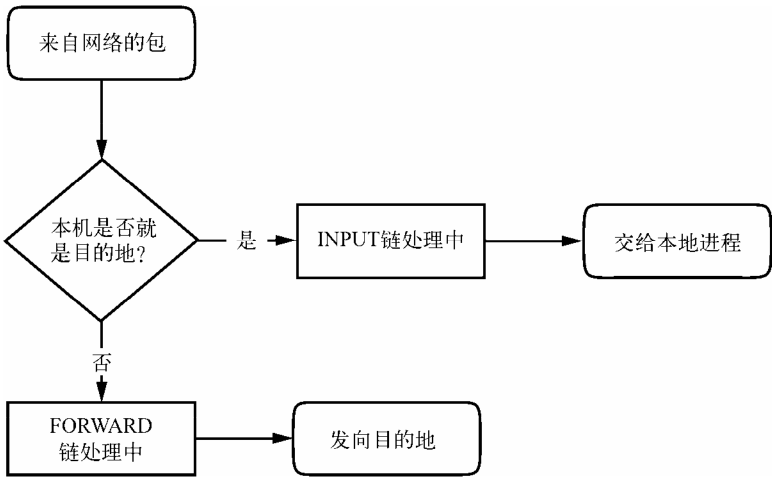


图9-5　接收包时经历的链



图9-6　发出包时经历的链

而实际上，整个过程并不只涉及这三个链。例如，PREROUTING和POSTROUTING链也会作用于数据包，而链进程也可能会在三个底层网络层次的任意一层发生。想看到全部的规则，可在网上搜“Linux netfilter packet flow”（Linux网络过滤器数据包流）。但这样搜出来的图，都尝试将数据包所有可能的路线纳入其中。而将路线图按数据包的来源进行分解会比较好理解，如图9-5和9-6所示。

9.21.2　设置防火墙规则

现在来看看IP表系统实际上是如何工作的。先用以下命令查看当前的设置：

```
# iptables -L
```

通常输出的链设置是空的：

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

如果没有指定对数据包使用何种规则，防火墙就会执行其默认的“策略”（policy）。上例中的三个链的默认策略都是ACCEPT，即让内核允许数据包通过。DROP策略是放弃数据包。命令iptables -P可以设置链的策略：

```
# iptables -P FORWARD DROP
```

警告　先别去管这些策略，把这一整节看完再下定论。

如果你对192.168.34.63这个用户不胜其烦，想将其屏蔽，你可执行这句：

```
# iptables -A INPUT -s 192.168.34.63 -j DROP
```

在上例中，参数-A INPUT给INPUT链增加了一个规则。而-s 192.168.34.63指定规则所针对的源IP，-j DROP指的是，当数据包符合规则时，内核会将其丢弃。因此，你的机器就会丢弃所有

来自192.168.34.63的数据包。

想查看设置好的规则，用iptables -L命令：

```
Chain INPUT (policy ACCEPT)
target     prot opt source             destination
DROP       all  --  192.168.34.63      anywhere
```

麻烦的是，192.168.34.63那家伙还发动了他子网上的所有人向你请求SMTP（TCP端口为25）连接。想摆脱他们，执行以下命令：

```
# iptables -A INPUT -s 192.168.34.0/24 -p tcp --destination-port 25 -j DROP
```

这个例子为源址增加了掩码限定符，并用-p tcp限定了只屏蔽TCP包。再详细一点，用--destination-port 25声明只屏蔽端口25。现在IP表中的INPUT应该是像下面这样：

```
Chain INPUT (policy ACCEPT)
target     prot opt source             destination
DROP       all  --  192.168.34.63      anywhere
DROP       tcp  --  192.168.34.0/24    anywhere            tcp dpt:smtp
```

现在问题解决了，但后来你会发现，你地址为192.168.34.37的朋友不能给你发邮件了，因为你连他也屏蔽了。为了快速修复，你可能会执行以下命令：

```
# iptables -A INPUT -s 192.168.34.37 -j ACCEPT
```

但不管用。要想查明原因，看看下面的新链：

```
Chain INPUT (policy ACCEPT)
target     prot opt source             destination
DROP       all  --  192.168.34.63      anywhere
DROP       tcp  --  192.168.34.0/24    anywhere            tcp dpt:smtp
ACCEPT     all  --  192.168.34.37      anywhere
```

内核是按从顶到底的顺序来读取链配置的，并执行第一条匹配的配置。

第一条并不匹配192.168.34.37，但第二条却可以，因为它适用于从192.168.34.1到192.168.34.254的所有主机，而第二条的策略是丢弃包。当某策略可用时，内核就按其执行，且不再往下看了。（你会发现192.168.34.37可以给你除了25外的所有端口发数据包，因为第二条规则只适用于端口25。）

解决方法是将该规则移到顶部。首先，执行以下命令来删掉第三条：

```
# iptables -D INPUT 3
```

然后，使用iptables -I在链的顶部插入第三条规则：

```
# iptables -I INPUT -s 192.168.34.37 -j ACCEPT
```

想在顶部以外的地方插入，可在链名后指定行号（如iptables -I INPUT 4 ...）。

9.21.3 防火墙策略

虽然以上教程告诉了你怎样插入规则，以及内核是如何处理IP链的，但我们还没真正了解防火墙是如何进行工作的。下面就来介绍一下。

防火墙的基本应用场景有两种：一种是保护单台机器（在每台机器的INPUT链插入规则），另一种是保护整个网络里的机器（在路由器的FORWARD链插入规则）。在这两种情况中，如果你只设置默认ACCEPT，并针对不断出现的垃圾信息数据包而持续添加DROP，那么就不能实现真正意义上的安全。你只能允许可信任的数据包流入，而对其他的数据包则进行阻断。

例如，如果你有一个SSH服务器在TCP端口22，那就不应该让别人连到你22以外的端口。要做到这样，首先可将INPUT链策略设置为DROP：

```
# iptables -P INPUT DROP
```

但可通过以下命令，允许ICMP通信（给ping或其他工具使用）：

```
# iptables -A INPUT -p icmp -j ACCEPT
```


确保你能收到自己发给自己的包，包括你自己的IP地址和localhost127.0.0.1。假设你IP地址为my_addr：

```
# iptables -A INPUT -s 127.0.0.1 -j ACCEPT
# iptables -A INPUT -s my_addr -j ACCEPT
```

如果你控制着你的整个子网（并信任其中所有东西），你可以将my_addr替换为你的子网地址和掩码，例如10.23.2.0/24。

现在，虽然你想阻止所有TCP连接请求，但你还得确保你能对外界发出TCP连接请求。而因为所有由外而内的TCP请求都是只有SYN位是被置位的，所以用以下这条命令可做到只屏蔽接收的请求：

```
# iptables -A INPUT -p tcp '!' --syn -j ACCEPT
```

接着，如果你用的是基于UDP的DNS，你必须允许接收域名服务器发给你的包，这样你的机器才能借助DNS查找域名。例如，允许接收/etc/resolv.conf中所有DNS服务器的数据包的话，用以下这条命令（ns_addr是域名服务器地址）：

```
# iptables -A INPUT -p udp --source-port 53 -s ns_addr -j ACCEPT
```

最后，允许所有SSH连接：

```
# iptables -A INPUT -p tcp --destination-port 22 -j ACCEPT
```

以上iptables设置适用于很多情况，涵盖了所有方向的连接（尤其是宽带），因为入侵者多喜欢逐个端口来攻击。你还可以按路由器的场景来改变一下防火墙的策略：把INPUT改成FORWARD，并在source和destination处填入子网。要进行更高级的设置，可以使用Shorewall之类的工具。

以上讨论仅与安全策略有关。其中关键的思想是，选出可信的，而不是选出可疑的。还有，IP防火墙只是网络安全的一部分而已。（下一章有更多介绍。）

9.22　以太网、IP 和 ARP

关于以太网上的IP协议，我们还有一点很有趣的没讲到。之前讲过，主机将IP数据包封装成以太网帧，以使其能在物理层上传输到另一台主机。我们还讲过，帧里面是不带有IP地址信息的，只有MAC地址。那么问题是：在为IP数据包进行帧的封装时，主机怎么知道哪个MAC对应哪个目标IP地址呢？

一般我们不考虑这个问题，因为网络软件包含了一套查找MAC地址的自动化系统，叫地址解析协议（Address Resolution Protocol，以下简称ARP）。使用以太网作为物理层、IP作为网络层的主机，维护着一个叫ARP缓存的表，其中包含了IP地址与MAC地址的映射关系。在Linux中，ARP缓存在内核中。想要查看你的机器的ARP缓存，可使用命令arp。（与其他网络命令类似，选项-n指定不进行DNS反向查询。）

```
$ arp -n
Address         Hwtype   Hwaddr              Flags Mask    Iface
10.1.2.141      ether    00:11:32:0d:ca:82   C             eth0
10.1.2.1        ether    00:24:a5:b5:a0:11   C             eth0
10.1.2.50       ether    00:0c:41:f6:1c:99   C             eth0
```

一台机器刚开机时，是没有ARP缓存的。那么MAC是怎样被缓存进来的呢？一切始于该机器想给另一台主机发送数据包时。如果目标IP不在ARP缓存中，则下列步骤会执行。

(1) 源主机创建一个包含ARP请求包的以太网帧，以求获知目标IP所对应的MAC。

(2) 源主机向目标子网的整个物理网络广播该帧。

(3) 如果子网中有个主机知道确切的MAC地址，它就会创建一个包含该地址的回复包和帧发给源主机。通常，发出回复的就是目标主机，它只是把自己的MAC地址回复给源主机。

(4) 源主机将IP-MAC地址对加入到ARP缓存中，并继续进行其他步骤。

注解　记住，ARP只应用于本地子网（可参考9.4节）。至于你子网以外的地方，你的主机只会将数据包发给路由器，之后的事就不由你管了。当然，你的主机还是要知道路由器的MAC的，而这也是用ARP来获取。

ARP唯一的实际问题是，如果你把某个IP的网络接口换了（例如在测试时），那么其缓存就相当于过期了，因为另一个网络接口的MAC是不同的。Unix系统会定时丢弃那些不活跃的ARP缓存条目，所以，除了那些过期的缓存数据会暂时带来一些小干扰外，并没有太大的问题。如果想立即删除的话，可以用这个命令：

```
# arp -d host
```

你还可以查看某个网络接口上的ARP缓存：

```
$ arp -i interface
```

arp(8)帮助手册介绍了如何手动设置ARP缓存，但你一般用不着这么做。

注解　别把ARP跟反向地址解析协议（Reverse Address Resolution Protocol，以下简称RARP）混淆。RARP按MAC地址来查找IP地址。在DHCP流行之前，一些无盘工作站和其他设备会用RARP来获取网络配置，而现在已经很少这样做了。

9.23　无线以太网

原理上，无线以太网（即WiFi）与有线网络没有太大区别。跟任何有线设备一样，它们都有MAC地址，也使用以太网帧来收发数据，因此Linux能像处理有线网络接口那样处理无线网络接口。网际层及其以上的层次都是一样的，主要不同就在于物理层中多加了一些组件，如频段、网络ID、安全组件等等。

无线网络的配置是十分开放的，不像有线设备那样拥有很强的自适应性。想要它正确地工作，Linux需要更多的配置工具。

下面快速浏览一下无线网络多出的组件。

- 传输细节：这些细节都是物理特性，例如频率。
- 网络标识：因为允许一套设备分出多个无线网络，所以必须要有办法区分它们。每个无线网络的标识就是它的服务集标识（Service Set Identifier，又称“网络名”，以下简称SSID）。
- 管理：虽然可以做到让无线网中的主机直接交流，但大多数做法还是通过无线接入点来沟通。无线接入点通常桥接着有线网络，使得它与有线网处于同一个网络中。
- 认证：你可能会想限制无线网的准入。想做到这点，你可以对接入点进行配置，以要求客户输入密码或其他认证码才能连上。
- 加密：除了限制连接请求，一般我们还会对电波进行加密以保信息安全。

这些组件的配置和工具分布在Linux的几个不同的地方。有些在内核：Linux有一套无线的扩展程序，以规范用户进程对硬件的访问。一旦讲到用户进程，那么无线网络的配置就变得复杂了，

所以大部分人更喜欢使用GUI的前端（例如NetworkManager的桌面应用）来做配置。尽管如此，我们还是来简单看一下无线网配置。

9.23.1　iw

你可以通过一个叫iw的工具来查看和改变内核空间设备与网络的配置。要使用它，你得先知道设备的网络接口名，例如本例的wlan0。以下是扫描可用无线网络的例子。（如果你是在市区，可能会显示大量可用无线网络。）

```
# iw dev wlan0 scan
```

注解　网络接口处于运行状态时，这条命令才有意义（网络接口没运行的话，请执行ifconfig wlan0 up），而网际层的参数（例如IP地址）则不用设置。

如果网络接口已接入到某个无线网络，那就可以这样来查看网络的详细情况：

```
# iw dev wlan0 link
```

其中的MAC地址是你所连的接入点的MAC地址。

注解　iw能够区分物理设备名（如phy0）和网络接口名（如wlan0），并允许你调整它们的各种设置。你甚至还可以给一个物理设备创建多个网络接口。然而，多数情况下，一般人都只会与网络接口名打交道。

以下命令用于将网络接口连接到一个不安全的无线网络：

```
# iw wlan0 connect network_name
```

能连接安全的网络固然最好。那么对于不太安全的有线等效加密协议（Wired Equivalent Privacy，以下简称WEP）来说，你可以在用iw的时候加上keys参数。然而，如果你很注重安全问题，那还是不要用WEP。

9.23.2　无线网络安全

Linux主要依靠一个叫wpa_supplicant守护进程来管理无线网络接口的认证和加密，以保证无线网络安全。这个守护进程可以处理WPA（WiFi Protected Access，WiFi网络安全接入）和WPA2的认证机制，以及几乎所有的无线网络加密技术。首次启动时，它会读取配置文件（默认是/etc/wpa_supplicant.conf），然后尝试在所指定的网络中向接入点提供自己的信息，并与其建立连

接。该系统的帮助文档很详尽，尤其是wpa_supplicant(1)和wpa_supplicant.conf(5)帮助手册中都有详细描述。

每次需要建立连接时才手动启动该守护进程，是很繁琐的。事实上，正是因为配置文件中选项很多，所以操作起来很乏味。更糟糕的是，`iw`和`wpa_supplicant`只是建立了物理层的连接，而没有配置好网际层的。所以你可以使用NetworkManager之类的自动化工具来解决问题。虽然这些工具也只是调用其他程序来工作，但是它们能正确地安排工作步骤，参考相应的配置文件，最终连好你的无线网络。

9.24　小结

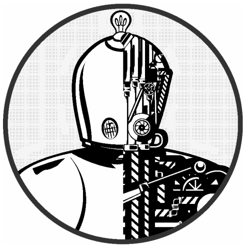
现在你应该理解了各网络层次的位置与角色，这对于理解Linux网络工作方式、实施网络配置是很关键的。虽然我们讲到的都是基础，但物理层、网际层、传输层的高级知识都能由此引申而来。所有层次都还可以分得更细，就像你刚才看到的无线网络的物理层还可以进一步细分。

本章讲解了大量针对内核的操作，涉及使用一些基本的用户空间控制工具修改内核数据结构（例如路由表）。这是操作网络的传统方式。然而，也像本书所提到的其他话题一样，出于灵活性的考虑，有些任务是不适合由内核来做的，于是我们使用用户空间的工具来做。其中还特别提到了用NetworkManager来监控和修改内核的配置。此外，一个例子就是动态路由协议，例如用在大型互联网路由器的边界网关协议（Border Gateway Protocol，以下简称BGP）。

现在你可能对网络配置有点厌倦了，下面我们谈谈使用网络吧，这就与应用层相关了。

第 10 章

网络应用与服务



本章探索基本的网络应用——运行在用户空间的应用层的客户端与服务器。因为该层在网络栈的顶层，靠近用户，所以你会觉得本章内容比上一章更好理解。这是当然的，因为你天天都使用网页浏览器或电子邮件阅读器之类的网络应用客户端。

客户端必须与它们相应的网络服务器连接起来才能正常工作。Unix服务器有很多种形式。服务器程序直接或间接地监听端口。另外，服务器功能各异，但没有通用的配置数据库。大多数服务器通过配置文件（尽管格式不统一）来定义自身的行为，并使用操作系统的syslog服务来记录日志。接下来我们会介绍一些常见的服务器以及用于调试的工具。

网络客户端使用操作系统的传输层协议与接口，所以我们必须对TCP和UDP传输层有基本的了解。下面就来看一些网络应用，先从一个使用TCP的网络客户端开始。

10.1　服务的基本概念

TCP服务是最好理解的概念之一，因为它建立在简单、无中断的双路数据流之上。或许最佳的解释方式，是让你直接通过TCP与某个Web服务器的80端口沟通，去看看数据是如何在该连接上移动的。例如，用以下命令连接一个Web服务器：

```
$ telnet www.wikipedia.org 80
```

你会得到类似这样的输出：

```
Trying some address...
Connected to www.wikipedia.org.
Escape character is '^]'.
```

现在输入：

```
GET / HTTP/1.0
```

敲两次ENTER，服务器就会回复一堆HTML文本，并终止本连接。

这个测试告诉我们：

- 远程主机那里有个Web服务器进程监听着TCP端口80；
- telnet是初始化这个连接的客户端。

注解　telnet原本是用于登录远程主机的。虽然非Kerberos的telnet是一种不安全的登录服务器（后面会讲到），但telnet客户端则是一个调试远程服务的有用工具。telnet只用到TCP，不会用到UDP或其他传输层协议。如果你要的是多用途网络客户端，可以考虑使用netcat，我们会在10.5.3节介绍它。

深入剖析

在上面的例子中，你用telnet手动与一个Web服务器进行了交互，使用到了应用层的协议HTTP。尽管你平时都是用网页浏览器来进行此类连接的，但我们还是只从telnet往前踏一小步，使用一个命令行工具来看看怎样与HTTP的应用层通信。我们要用的是curl，以及一个记录通信细节的选项：

```
$ curl --trace-ascii trace_file http://www.wikipedia.org/
```

注解　你的发行版可能没有预装curl包，但安装该包对你来说应该不是难事。

你会得到大量的HTML输出。忽略它们（或将它们重定向到/dev/null），并去看看刚刚创建出来的文件trace_file。假设连接成功，那么文件的开头部分看起来应该是下面这样的，表明刚开始时curl尝试跟该服务器建立TCP连接：

```
== Info: About to  connect() to www.wikipedia.org port 80 (#0)
== Info:   Trying  10.80.154.224... == Info: connected
```

至此你看到的一切都是发生在传输层或其下层的。然而，如果连接成功了，那么curl就会尝试发送请求（即“报头”）；这里就开始有应用层了。

```
=> Send header, 167 bytes (0xa7)
0000: GET / HTTP/1.1
0010: User-Agent: curl/7.22.0 (i686-pc-linux-gnu) libcurl/7.22.0 OpenS
```

```
0050: SL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp/2.3
007f: Host: www.wikipedia.org
0098: Accept: */*
00a5:
```

上例第一行是curl的调试信息输出，告诉了你它接下来要做的事情。剩下的行展示了curl发送给服务器的信息。粗体的内容是发送到服务器的。开头的十六进制数只是一个调试偏移量，告诉你收发的数据有多少。

你可以看到curl先发了一个GET命令给服务器（就像用telnet那样），接着是一些有关服务器的额外信息以及一个空行。然后服务器回应了，首先是它的报头（粗体部分）。

```
<= Recv header, 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header, 16 bytes (0x10)
0000: Server: Apache
<= Recv header, 42 bytes (0x2a)
0000: X-Powered-By: PHP/5.3.10-1ubuntu3.9+wmf1
--snip--
```

跟之前那段输出很像，<=开头的行是调试信息，0000:是偏移量。

服务器回应的报头可以很长，然后，某行的报头出现了我们请求的内容，就像下面这样：

```
<= Recv header, 55 bytes (0x37)
0000: X-Cache: cp1055 hit (16), cp1054 frontend hit (22384)
<= Recv header, 2 bytes (0x2)
0000:
<= Recv data, 877 bytes (0x36d)
0000: 008000
0008: <!DOCTYPE html>.<html lang="mul" dir="ltr">.<head>.<!-- Sysops:
--snip--
```

该输出同样展示了应用层的一个重要特性。尽管调试信息里有Recv header和Recv data，意味着服务器返回的信息可分为两种，但curl向操作系统获取这两种信息的方法却没有不同，操作系统对它们的处理方式也没有不同，甚至底层网络对它们的数据包的处理方式也是一样的。如果说有不同，那完全是在用户空间的curl内部。curl读取报头的时候，如果发现了标志HTTP报头结束的空行（即中间的两个字节），它就知道接下来将是真正的应答内容了。

在服务器发送数据方面也同样如此。服务器并不区分发给操作系统的报头和内容，区分只发生于用户空间的服务器程序。

10.2　网络服务器

大多数网络服务器跟cron之类的服务器守护进程很像，只不过网络服务器是与网络端口进行交互的。事实上，我们在第7章讲到的syslogd，它如果加上-r选项的话，就会接受514端口的UDP数据包。

以下是一些常见的网络服务器，在你系统中可能也会见到。

- httpd、apache、apache2：Web服务器。
- sshd：Secure shell守护进程（见10.3节）。
- postfix、qmail、sendmail：邮件服务器。
- cupsd：打印服务器。
- nfsd、mountd：网络文件系统（文件共享）守护进程。
- smbd、nmbd：文件共享守护进程（见第12章）。
- rpcbind：远程程序调用（RPC）端口映射服务守护进程。

大多数网络服务器有一个共同的特性，即它们通常是多进程的。其中至少有一个进程在监听网络端口，而当它接收到一个新的连接时，就会使用fork()来创建一个子进程，负责那个新的连接。该子进程，也叫*辅助进程*，会随着连接的终止而终止。同时，监听进程会继续接收连接。这样，一个服务器就能轻松地处理多个连接，一般不会有什么问题。

然而，这个模型也会有一些异常情况。调用fork()是会增加系统负担的，而高性能TCP服务器（如Apache Web服务器）能在启动时就创建一定数量的辅助进程，以备连接需要。接受UDP包的服务器只会简单地接收数据并对其做出反应，它们不需要维持连接。

10.3 SSH

各种服务器的工作方式都是有些许的差别的。现在我们来详细看一下独立的Secure shell（以下简称SSH）服务器。SSH是最常见的网络服务应用之一，它是一种远程连接Unix机器的标准。配置好之后，我们就能通SSH进行安全的shell登录、执行远程程序、共享简单的文件等。此外，SSH还凭借公钥认证和简单的会话加密，取代了旧的、不安全的远程登录系统telnet和rlogin。大多数ISP和云提供商都要求以SSH来使用他们的服务，另外很多基于Linux的网络设备（如NAS）也是这样要求的。OpenSSH（http://www.openssh.com/）是一个比较流行的、针对Unix的SSH实现，而且几乎所有的Linux发行版也预装了它。OpenSSH的客户端是ssh，服务器是sshd。SSH协议有两个主要版本：1和2。OpenSSH对两者都支持，但1是很少用的。

SSH的功能和特性使它能做到以下事情。

- 对密码和会话内容加密，保护你不受窃听困扰。
- 作为其他网络连接的管道，包括来自X Window客户端的连接。X会在第14章详细介绍。
- 几乎所有操作系统都可用SSH连接。
- 使用密钥做主机认证。

注解 作为其他网络连接的管道的意思是，使用一个进程将某个连接包装并转换成另一个。采用SSH包装X Window连接的好处是，它在提供显示环境的同时，还对X的数据进行了加密。

但SSH也有缺点。其中一个就是，若想建立SSH连接，你必须先知道远程主机的公钥，它是

不需要通过什么保密的渠道就能获得的（当然你也可以手动检查它的真假）。想知道那些加密技术的运作方式，可参考Bruce Schneier的*Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd edition（Wiley，1996）。深入介绍SSH的书有两本：Michael W. Lucas的*SSH Mastery: OpenSSH, PuTTY, Tunnels and Keys*（Tilted Windmill Press，2012），以及Daniel J. Barrett、Richard E. Silverman和Robert G. Byrnes的*SSH*, *The Secure Shell*, 2nd edition（O'Reilly，2005）。

10.3.1　SSHD服务器

运行sshd需要一个配置文件以及主机密钥。大多数发行版都将配置文件放在/etc/ssh配置目录中，并在你安装它们的sshd包时，尝试将一切都配置好。（这里的配置文件名sshd_config跟客户端的文件名ssh_config很容易混淆，请注意区分。）

你应该不用改变sshd_config里的任何东西，但检查一下是可以的。这个文件包含了键值对，如下列片段所示：

```
Port 22
#Protocol 2,1
#ListenAddress 0.0.0.0
#ListenAddress ::
HostKey /etc/ssh/ssh_host_key
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
```

以#开头的是注释，而sshd_config中的很多的注释都暗示了默认值。sshd_config(5)手册包含了所有可选值的解释，而以下这些是最重要的。

- `HostKey file`：使用`file`作为主机密钥。（主机密钥是短的。）
- `LogLevel level`：按照syslog的级别`level`来记录信息。
- `PermitRootLogin value`：`value`为yes则允许超级用户通过SSH登录，否则不允许。
- `SyslogFacility name`：按syslog设施的名字`name`来记录信息。
- `X11Forwarding value`：若`value`为yes则允许X Window客户端被SSH管道接驳。
- `XAuthLocation path`：设置`xauth`的路径。找不到路径的话，X11的SSH管道将无法工作。如果`xauth`不在/usr/bin里，则需写明`xauth`的完整路径。

主机密钥

OpenSSH有三套主机密钥：一套用于版本1的协议，另两套用于版本2。每套都有一个公钥（扩展名为.pub的文件）和一个私钥（无扩展名）。不要让任何人知道你的私钥，否则就有被入侵的危险。

SSH版本1只有RSA密钥，而版本2则有RSA和DSA。RSA和DSA都是公钥加密算法。密钥的文件名如表10-1所示。

表10-1 OpenSSH密钥文件

文件名	密钥类型
ssh_host_rsa_key	RSA 私钥（版本2）
ssh_host_rsa_key.pub	RSA 公钥（版本2）
ssh_host_dsa_key	DSA 私钥（版本2）
ssh_host_dsa_key.pub	DSA 公钥（版本2）
ssh_host_key	RSA 私钥（版本1）
ssh_host_key.pub	RSA 公钥（版本1）

一般你不用自己建立密钥，因为OpenSSH的安装程序或你发行版的安装脚本会为你做好，但若你要使用ssh-agent之类的程序，你还是需要知道密钥是如何创建的。比如创建SSH版本2的密钥，可用OpenSSH自带的ssh-keygen程序：

```
# ssh-keygen -t rsa -N '' -f /etc/ssh/ssh_host_rsa_key
# ssh-keygen -t dsa -N '' -f /etc/ssh/ssh_host_dsa_key
```

版本1则是：

```
# ssh-keygen -t rsa1 -N '' -f /etc/ssh/ssh_host_key
```

SSH服务器与客户端还用到了一个叫作ssh_known_hosts的密钥文件，里面包含了其他主机的公钥。如果你要使用基于主机的认证，服务器端的ssh_known_hosts必须要包含所有可信客户端的公钥。了解密钥文件有助于你更换机器。当你从头给机器安装系统时，你可以用回旧的机器上的密钥文件，这样用户就能跟旧的密钥配置对上了。

开启SSH服务器

尽管大多数发行版都带有SSH，但默认却不会启动sshd服务器。在Ubuntu和Debian上，安装SSH服务器就会创建密钥、启动服务，并在开机脚本中加上启动SSH。在Fedora上，sshd是预装的，但默认是关闭的。想在开机时启动sshd，可以用chkconfig（但这并不会让服务器马上启动，除非你用service sshd start）：

```
# chkconfig sshd on
```

Fedora一般会在sshd初次启动时补建漏掉的密钥文件。

如果你还没装好init的支持，就用root来运行sshd，以启动服务器。而一旦启动，sshd就会在/var/run/sshd.pid记下自己的PID。

你还可以在systemd或用inetd以套接字单元来启动sshd，但这不是个好方法，因为服务器偶尔会产生密钥文件，这个进程可能会很费时间。

10.3.2 SSH客户端

想登录远程主机，运行：

```
$ ssh remote_username@host
```

如果你在本机的账号跟远程的一样，可以省略remote_username@。你还可以像下面例子一样用管道符来连接ssh命令，将一个叫dir的目录复制到另一台主机：

```
$ tar zcvf - dir | ssh remote_host tar zxvf -
```

全局的SSH客户端配置文件ssh_config应该在/etc/ssh里，就如sshd_config文件一样。跟服务器配置文件类似，客户端配置文件也有键值对，但你应该不用去改它。

SSH客户端的最常见问题是，你本机的ssh_known_hosts或.ssh/known_hosts里的公钥跟远程主机的不匹配，这会导致如下报错：

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
38:c2:f6:0d:0d:49:d4:05:55:68:54:2a:2f:83:06:11.
Please contact your system administrator.
Add correct host key in /home/user/.ssh/known_hosts to get rid of this message.
Offending key in /home/user/.ssh/known_hosts:12 ❶
RSA host key for host has changed and you have requested strict checking.
Host key verification failed.
```

通常这表示远程主机的管理员更改了密钥（经常是在更换硬件时发生）。为了确认状况，不妨跟管理员联系一下。总之，以上的信息告诉你，错误的密钥是在用户的.ssh/known_hosts文件的第12行，如上例中❶处所示。

如果你确定没问题，那就移除错误的那行，或换一个正确的公钥。

SSH文件传输客户端

OpenSSH包含了文件传输程序：scp和sftp。这两个命令用以取代旧的、不安全的命令rcp和ftp。

你可以用scp在本机和远程主机之间传输文件，它就像cp命令一样。以下是一些例子。

```
$ scp user@host:file .
$ scp file user@host:dir
$ scp user1@host1:file user2@host2:dir
```

sftp程序就好比是命令行的ftp客户端，它有get和put命令。远程主机必须装好sftp-server程序（如果远端装了OpenSSH，那通常都会带上这个）。

注解 如果你对功能性和灵活性的要求超越了scp和sftp所能提供的（例如你经常进行大宗文件的传送），那就试下rsync，这个命令会在第12章讲到。

非Unix平台的SSH客户端

所有流行的操作系统都有相应的SSH客户端，正如OpenSSH网站上列出的（http://www.openssh.com/）。你要选哪个呢？PuTTY是一个不错的基本Windows客户端，它包含了安全的文件复制程序。MacSSH适用于Mac OS 9.x及以下版本。Mac OS X是基于Unix的，也包含OpenSSH。

10.4　守护进程 inetd 和 xinetd

为每种服务实现独立的服务器好像并不太高效。每个服务器都要分别配置端口监听、访问控制以及端口设置。大多数服务的配置方式都是一样的，只是处理连接的方式不同。

传统的解决方法是使用inetd守护进程，它是一种*超级服务器*，用于规范网络端口的接入和服务器程序与网络端口之间的接口。启动inetd之后，它会读取自己的配置文件，并监听其中提到的网络端口。当连接到来时，inetd就会新开一个进程来处理它。

xinetd是inetd的新版本，它提供更简单的配置和更优秀的访问控制，但xinetd正被systemd取代，因为如6.4.7节所述，systemd的套接字单元能提供同样的功能。

尽管inetd将被淘汰，但它的配置能告诉我们建立服务的要素。如下列配置文件/etc/inetd.conf所示，`sshd`也能被inetd发起，而不只作为一个独立的服务器存在：

```
ident       stream    tcp    nowait    root    /usr/sbin/sshd    sshd -i
```

其中7个字段从左至右含义分别如下所示。

- 服务名称：来自/etc/services（见9.14.3节）。
- 套接字类型：通常TCP是`stream`，UDP是`dgram`。
- 协议：传输协议，通常是`tcp`或`udp`。
- 数据报服务器行为：UDP可选`wait`或`nowait`。其他传输协议应该用`nowait`。
- 用户：运行该服务的用户。加上`.group`的话，可以指定群组。
- 可执行程序：inetd为应付该服务而发起的程序。
- 参数：可执行程序的参数。第一个参数是该程序的名字。

TCP封装器：tcpd、/etc/hosts.allow和/etc/hosts.deny

在低层次防火墙流行之前，很多管理员都用TCP封装器的库和守护进程来操控网络服务。具体做法是，让inetd运行`tcpd`程序，而`tcpd`在接收连接时会查看/etc/hosts.allow和/etc/hosts.deny文件中的访问控制列表。`tcpd`会记下该连接，并在审查通过后，将连接交给最终的服务程序。（虽然仍有系统采用TCP封装器，但因为它的使用率已不高，所以我们不会详细介绍。）

10.5　诊断工具

下面来看下一些比较有用的针对应用层的诊断工具。它们有些还会涉及传输层和网际层，因

为应用层最终还是需要这些底层支撑。

如第9章讲到的，netstat是一个基本的网络服务调试工具，能显示一些传输层和网际层的统计信息。表10-2罗列了一些考察连接的有用选项。

表10-2　一些有用的netstat报告选项

选　项	描　述
-t	打印TCP端口信息
-u	打印UDP端口信息
-l	打印监听中的端口
-a	打印所有活动中的端口
-n	取消域名查找（能加快速度，就算没使用DNS也有效）

10.5.1　lsof

在第8章我们讲过，lsof能够跟踪打开的文件，但其实它还可以列出正在使用或监听端口的程序。想完整列出这些程序，可以运行以下命令：

```
# lsof -i
```

若以普通用户身份运行，它只会显示出该用户的进程。而以root身份运行的话，就会列出全部进程，像下面这样：

```
COMMAND     PID     USER    FD   TYPE   DEVICE  SIZE/OFF NODE NAME
rpcbind     700     root    6u  IPv4    10492       0t0  UDP *:sunrpc
rpcbind     700     root    8u  IPv4    10508       0t0  TCP *:sunrpc(LISTEN)
avahi-dae   872    avahi   13u  IPv4 21736375       0t0  UDP *:mdns
cupsd      1010     root    9u  IPv6 42321174       0t0  TCP ip6-localhost:ipp (LISTEN)
ssh       14366    juser    3u  IPv4 38995911       0t0  TCP thishost.local:55457->
    somehost.example.com:ssh (ESTABLISHED)
chromium- 26534    juser    8r  IPv4 42525253       0t0  TCP thishost.local:41551->
    anotherhost.example.com:https (ESTABLISHED)
```

这个例子展示了服务器程序和客户端程序的用户和PID：从顶部的旧式RPC服务，到avahi提供的组播DNS服务，甚至还有兼容IPv6的打印机服务（cupsd）。最后两条是客户端连接：一个SSH连接和一个Chromium网页浏览器。因为输出可能会很长，所以最好是加上过滤器（下面会讲到）。

lsof与netstat相似的一点是，它试图将每个IP反向解析成主机名，这会减慢结果的输出。我们可以使用-n选项来阻止它这么做：

```
# lsof -n -i
```

你还可以用-P取消对/etc/services的端口名查找。

按协议和端口来过滤

如果你正在寻找一个特定的端口（假设你知道有个进程在使用该端口，而你想知道是哪个进

程），那就执行：

```
# lsof -i:port
```

完整的语法如下：

```
# lsof -iprotocol@host:port
```

protocol、@host、和:port参数都是可选的，它们会对lsof的输出进行过滤。就如大多数的网络工具一样，host和port可以填名字或数字。例如你只想查看TCP端口80（HTTP端口）的连接，就用：

```
# lsof -iTCP:80
```

按连接状态来过滤

lsof还有个特别好用的过滤器——连接状态。例如，只查看正在监听TCP端口的进程，就用：

```
# lsof -iTCP -sTCP:LISTEN
```

这个命令能让你概览运行中的网络服务器进程。然而，因为UDP服务器并不维持连接，所以你要用-iUDP来查看服务器和客户端。通常这不是什么问题，因为一般你系统中不会有太多UDP服务器。

10.5.2 tcpdump

如果你想明确地知道你的网络上有什么在流通，你可用tcpdump将网络接口置于混杂模式，并向你报告每一个通过的数据包。如下所示，不带参数地运行tcpdump，就包含了ARP请求和Web连接。

```
# tcpdump
tcpdump: listening on eth0
20:36:25.771304 arp who-has mikado.example.com tell duplex.example.com
20:36:25.774729 arp reply mikado.example.com is-at 0:2:2d:b:ee:4e
20:36:25.774796 duplex.example.com.48455 > mikado.example.com.www: S
3200063165:3200063165(0)  win  5840  <mss  1460,sackOK,timestamp38815804[|tcp]>
(DF)
20:36:25.779283 mikado.example.com.www > duplex.example.com.48455: S
3494716463:3494716463(0)  ack  3200063166  win  5792  <mss1460,sackOK,timestamp
4620[|tcp]> (DF)
20:36:25.779409 duplex.example.com.48455 > mikado.example.com.www: .ack 1 win
5840 <nop,nop,timestamp 38815805 4620> (DF)
20:36:25.779787 duplex.example.com.48455 > mikado.example.com.www: P 1:427(426 )
ack 1 win 5840 <nop,nop,timestamp 38815805 4620> (DF)
20:36:25.784012 mikado.example.com.www > duplex.example.com.48455: .ack 427
win 6432 <nop,nop,timestamp 4620 38815805> (DF)
20:36:25.845645 mikado.example.com.www > duplex.example.com.48455: P 1:773(772)
ack 427 win 6432 <nop,nop,timestamp 4626 38815805> (DF)
```

```
20:36:25.845732 duplex.example.com.48455 > mikado.example.com.www: .ack 773
win 6948 <nop,nop,timestamp 38815812 4626> (DF)

9 packets received by filter
0 packets dropped by kernel
```

你可以加入过滤器，以让tcpdump的内容更精确。过滤器可以是源主机、目标主机、网络、以太网地址、各层协议等等。各种数据包协议中，tcpdump能认出的有：ARP、RARP、ICMP、TCP、UDP、IP、IPv6、AppleTalk、IPX。例如，让tcpdump只输出TCP数据包，可运行：

```
# tcpdump tcp
```

想看网页包和UDP包，运行：

```
# tcpdump udp or port 80
```

注解　如果是需要进行大量的数据包嗅听，可用诸如Wireshark类的GUI工具来代替tcpdump。

表达元

在上面的例子中，tcp、udp和port 80，都是表达元。表10-3列举了最重要的一些表达元。

表10-3　tcpdump表达元

表达元	指定包
tcp	TCP包
udp	UDP包
port *port*	来自（去往）端口*port*的TCP包和（或）UDP包
host *host*	来自（去往）主机*host*的包
net *network*	来自（去往）网络*network*的包

运算符

上例中的or是一个运算符。tcpdump有很多种运算符（例如and和!），并且可以将运算符括起来。如果你要用tcpdump来做一些复杂的工作，请仔细阅读它的帮助手册，尤其是关于表达元的那一节。

什么时候不该用tcpdump

使用tcpdump时要格外留心。本节前面的例子中只打印出了TCP（传输层）和IP（网际层）的报头的信息，但你可以指定tcpdump打印出整个数据包的内容。尽管很多网络工具都有这样的功能，但你最好是不要随便嗅听网络，除非该网络是你所有的。

10.5.3　netcat

如果你觉得用telnet host port连接远程主机不够灵活，试试netcat（或nc）。netcat可以与TCP和UDP端口通信，指定本地端口，监听端口，扫描端口，对网络输入输出重定向到标准输入

输出等等。用netcat连接TCP端口，如下所示：

```
$ netcat host port
```

netcat只在对方关闭连接时终止，所以如果你用netcat接收标准输入，就会有点麻烦。你可以使用CTRL-C来随时关闭连接。（如果你希望由标准输入流来决定程序和网络连接的关闭，可以改用sock程序。）

想监听某个端口，可执行以下命令：

```
$ netcat -l -p port_number
```

10.5.4　扫描端口

有时你可能想知道你网络中的机器正提供着什么服务，或哪个IP正在被使用，这时你可以用网络映射器（Network Mapper，以下简称Nmap）程序来扫描并列举出一台机器（或一个网络中的机器）的开放端口。大多数的发行版都有自己的Nmap包，或者你可以从http://www.insecure.org/获取。（想了解Nmap的功能，请看帮助手册及在线资源。）

在列举你机器的端口时，至少从这样两个不同的角度来运行Nmap程序会更好：从本机和从另一台机器（可能是本地网络之外的）。这样做可以看到你防火墙屏蔽了什么。

警告　如果你想用Nmap扫描的网络是由他人控制的，那就需要获得准许。网络管理员通常会监察这种端口扫描，并屏蔽那些发出扫描的机器。

运行**nmap** ***host***来对一个主机进行端口扫描，例如：

```
$ nmap 10.1.2.2
Starting Nmap 5.21 ( http://nmap.org ) at 2015-09-21 16:51 PST
Nmap scan report for 10.1.2.2
Host is up (0.00027s latency).
Not shown: 993 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
25/tcp    open  smtp
80/tcp    open  http
111/tcp   open  rpcbind
8800/tcp open  unknown
9000/tcp open  cslistener
9090/tcp open  zeus-admin

Nmap done: 1 IP address (1 host up) scanned in 0.12 seconds
```

如你所见，这里有很多服务开启了，其中不少服务是多数linux版本默认关闭的。事实上，通

常是默认开启的就只有111端口，即rpcbind端口。

10.6　远程程序调用

上节提到的rpcbind服务是什么意思呢？RPC意指远程程序调用（Remote Procedure Call），是应用层中较低层的一个系统。它是为了方便程序员访问网络应用而设计的，能使本地程序调用远程程序（按程序号来标识），让远程程序返回结果码或信息。

RPC的实现需要用到传输层协议，如TCP和UDP。它需要一种特别的中介服务来将程序号与TCP和UDP的端口号对应起来。这种服务就是rpcbind，运行RPC服务就需要用到它。

想看你计算机运行了什么RPC服务，可执行：

```
$ rpcinfo -p localhost
```

RPC是一种难以消亡的协议。网络文件系统（Network File System，以下简称NFS）和网络信息服务（Network Information Service，以下简称NIS）就用得到它，但单机环境下是不需要这些服务的。当你以为你已经对rpcbind不会再有任何需要时，就会有些依赖它的东西出现，例如GNOME的文件访问监控（File Access Monitor，以下简称FAM）。

10.7　网络安全

因为Linux是PC平台上风行的Unix系统，尤其是它被广泛用作网页服务器，所以它也招致了很多黑客攻击。9.21节已介绍过了防火墙，但网络安全的话题不止那些。

网络安全带来了两个极端：真心想攻破系统的人（不管是为了钱还是为了好玩）和精心设计防御方案的人（这个也很有利可图）。幸好，你不需要了解太多东西来维持系统的安全。这里有一些经验法则。

- 打开的服务越少越好：黑客不能攻击你机器上不存在的服务。如果你知道有哪个服务是你不需要的，那就别打开它，等你真正用到那个服务时再打开。
- 防火墙屏蔽得越多越好：Unix系统有一些内部服务是你可能不知道的（例如用于RPC端口映射的TCP端口111），因此也别让其他机器知道它们的存在。跟踪和规范本机的服务是很困难的，因为监听的程序和被监听的端口错综复杂。为避免黑客发现你系统上的内部服务，请使用有效的防火墙规则，并为路由器安装防火墙。
- 记录你提供给互联网的服务：如果你运行着SSH服务器、Postfix或类似的东西，请保持你软件的更新，以及使用合适的安全警报（详见10.7.2节）。
- 让服务器使用“长期支持”的系统版本：安全团队一般在稳定、有支持的系统版本上才能集中精力工作。开发版或测试版如Debian Unstable和Fedora Rawhide不太受他们关注。
- 账号不要发给不用的人：通过本地账号获取超级用户的权限，比远程入侵要简单得多。事实上，大多数系统上的软件都有这样那样的漏洞，只要你能登录shell，就有可能获取超

级用户的权力。不要指望你的朋友懂得如何保护密码（或懂得使用复杂的密码）。

- 避免安装可疑的二进制包：它们可能包含木马。

进行安全保护的方法基本就这些。但为什么要这么做呢？因为有以下三种基本的网络攻击。

- 全面威胁：意思是获取了超级用户的权限（对一台机器完全掌控）。黑客可以通过服务攻击，例如缓存溢出、接管一个没什么保护措施的账号或利用一个写得不太好的setuid，来做到“全面威胁”。
- 拒绝服务（Denial-of-service，简称DoS）攻击：这使得机器无法继续提供服务，或无需通过登录就用某些方法造成机器故障。这种攻击更难防范，但很容易应对。
- 恶意软件：Linux用户大多对恶意软件（如电子邮件蠕虫和病毒）免疫，只因为他们的电子邮件客户端不会蠢到运行附件里的程序。但Linux恶意程序确实存在。请勿从陌生的地方下载并安装二进制软件。

10.7.1　典型漏洞

有两种重要的漏洞是需要担心的：直接攻击和嗅探明文密码。所谓直接攻击，就是摆明要攻陷一台机器。最常见的做法是利用缓存溢出。该漏洞是因为粗心的程序员没检查数组边界而造成的。攻击者在一大堆数据中编造一个栈帧，然后送到远程服务器，并期望它溢出而覆盖了程序，最终导致服务器执行了栈帧中的东西。尽管这不太容易，但却很容易复制。

第二种漏洞，即嗅探明文密码，是指获取网络上明文传输的密码。一旦攻击者拿到你的密码，那你就玩完了。从那开始，攻击者就会直接登入，尝试获取超级用户的权限（这比远程攻击要简单），或以该机器攻击其他机器，或用其他机器攻击该机器。

10

> 注解　如果你有些服务是不带加密功能的，那就试试Stunnel（http://www.stunnel.org/）。它是一个加密封装器数据包，就像TCP封装器一样。跟`tcpd`一样，Stunnel擅长封装inetd服务。

有些服务因为设计缺陷而经常成为攻击目标。例如以下这些服务，你应该停用它们（其实大多数系统都默认关闭的）。

- `ftpd`：不管什么原因，所有的FTP服务器都存在漏洞。除此之外，大多数FTP服务器都用明文密码。如果你要在机器之间移动文件，考虑下基于SSH的方案或`rsync`服务器。
- `telnetd`、`rlogind`、`rexecd`：它们都把会话内容（包括密码）用明文传输。不要使用它们，除非你有Kerberos版本。
- `fingerd`：黑客可以通过finger服务获取用户列表和其他信息。

10.7.2　安全资源

以下是三个不错的安全网站。

- http://www.sans.org/：提供培训、服务、免费的高危漏洞周报、安全策略样板等等。
- http://www.cert.org/：这里介绍最危险的漏洞。
- http://www.insecure.org/：这里有Nmap和各种网络开发测试工具，比其他网站做得更详尽、更开放。

如果你对网络安全有兴趣，你应该学习一下传输层安全（Transport Layer Security，TLS）及其前身——SSL（安全套接层）。这些用户空间的层次被加插到客户端和服务器，用公钥加密和认证来支援网络通信。Davies的*Implementing SSL/TLS Using Cryptography and PKI*（Wiley，2011）是本不错的指南。

10.8　前瞻

如果你对一些复杂的网络服务器感兴趣，这里有两个常见的：Apache网页服务器和Postfix邮件服务器。尤其是Apache，它安装简单，而且大多数系统版本都有它的安装包。如果你的机器装有防火墙或NAT路由器，你可以不用担心安全问题，尽情体验。

在前面的几章里，我们逐渐从内核空间过渡到了用户空间。本章只有少量工具是与内核交互的，如`tcpdump`。本章余下的部分讲解套接字如何将内核的传输层与用户空间的传输层接合起来。这是一个进阶内容，程序员可能比较感兴趣，所以无兴趣的话可跳到下一章。

10.9　套接字：进程与网络的通信方式

现在我们稍微转变一下话题，看看进程是怎样从网络读数据，以及怎样向网络写数据的。向建立好的连接读写数据是很容易的，只需要做一些系统调用（参考recv(2)和send(2)帮助手册）工作。从进程的角度来看，或许最重要的是要知道在做系统调用时如何与网络对应。在Unix上，进程是用套接字来标识与网络通信的时机与方式的。套接字是进程通过内核访问网络的接口，它代表用户空间与内核空间的边界。它也常被用于进程间通信（Interprocess Communication，以下简称IPC）。

因为进程需要以不同的方式访问网络，所以套接字也分不同种类。例如，TCP连接会用流式套接字（程序员称之为`SOCK_STREAM`），而UDP连接则用数据报套接字（`SOCK_DGRAM`）。

建立网络套接字有点复杂，因为你有时需要知道套接字类型、IP地址、端口、传输协议。然而，当所有初始的细节整理好后，服务器就能用相应的标准模型来处理网络通信了。

图10-1的流程展示了服务器处理收到的流式套接字连接。注意，这种服务器涉及到两种套接字：监听套接字和读写套接字。主进程用监听套接字在网络中寻找连接。当一个新的连接到来，主进程就用`accept()`系统调用来接收该连接，它能为连接创建专用的读写套接字。接着主进程用`fork()`创建一个新的子进程来处理该连接。最终，监听套接字继续监听，为主进程带来更多连接。

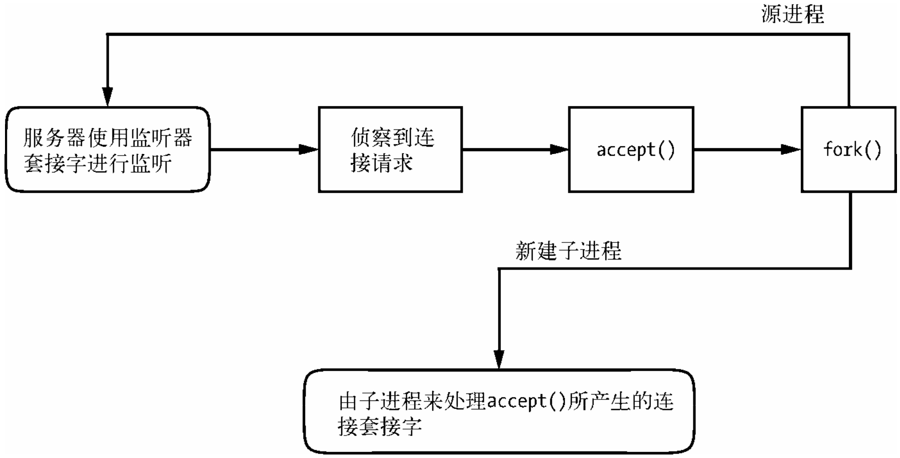


图10-1 接受和处理新连接的一种方法

如果你是个程序员，并且想知道如何使用套接字接口，可看一下W. Richard Stephens、Bill Fenner和Andrew M. Rudoff所著*Unix Network Programmin*, Volume 1, 3rd edition（Addison-Wesley Professional，2003），这是一本经典教程，第二卷也讲到了进程间通信。

10.10 Unix 域套接字

使用网络设施的应用不需要将两台分隔的主机牵扯在一起。很多应用都是用像客户端-服务器或点对点那样的机制建立起来的，同一台机器上的进程则使用IPC（进程间通信）来协商有哪些工作要做，以及由谁来做。例如，回想一下systemd和NetworkManager这样的守护进程利用D-Bus来监控和应对系统事件的做法。

进程间的通信可以使用本地主机（127.0.0.1）来做常规的网络通信，但我们一般使用另一种特别的套接字，这种套接字在第三章中简单提到过，叫*Unix域套接字*。进程与Unix域套接字的连接几乎与网络套接字连接一样：进程可以监听和接收套接字上的连接，还可以选择不同类型的套接字来实现TCP式或UDP式的工作方式。

注解 要记住，Unix域套接字并不是网络套接字，其背后也没有网络。使用它不需要配置网络。而且，Unix域套接字不需要非得与套接字文件绑定。进程可以创建非命名Unix域套接字，并与其他进程分享它的地址。

10.10.1 对开发者的好处

开发者们喜欢Unix域套接字的以下两点。第一，开发者可以通过管理套接字文件的访问权限来管理Unix域套接字的访问权限。也就是说，不能访问某个套接字文件的进程，也就不能使用该

套接字。此外，因为它没有网络交互，所以更简单，更能减少网络攻击的机会。例如，你可以在/var/run/dbus里找到D-Bus的套接字文件：

```
$ ls -l /var/run/dbus/system_bus_socket
srwxrwxrwx 1 root root 0 Nov 9  08:52 /var/run/dbus/system_bus_socket
```

第二，因为Linux内核与Unix域套接字通信并不需要经历网络层次，所以性能会比网络套接字好。

为Unix域套接字写代码跟支持一般的网络套接字没多大不同。因为它的好处十分明显，所以有些网络服务器会同时提供网络套接字和Unix域套接字的连接。例如，MySQL的数据库服务器`mysqld`能接受远端的连接，同时也以/var/run/mysqld/mysqld.sock提供Unix域套接字。

10.10.2　列出Unix域套接字

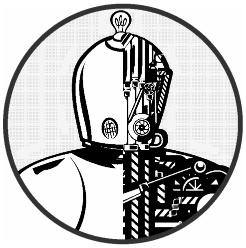
你可以使用`lsof -U`来查看系统正在使用中的Unix域套接字：

```
# lsof -U
COMMAND      PID     USER    FD   TYPE      DEVICE  SIZE/OFF     NODE NAME
mysqld     19701    mysql   12u   unix 0xe4defcc0       0t0 35201227 /var/run/mysqld/mysqld.sock
chromium-  26534    juser    5u   unix 0xeeac9b00       0t0 42445141 socket
tlsmgr     30480  postfix    5u   unix 0xc3384240       0t0 17009106 socket
tlsmgr     30480  postfix    6u   unix 0xe20161c0       0t0    10965 private/tlsmgr
--snip--
```

因为很多现代的应用都用到了非命名套接字，所以这个列表可能会很长。你可以借助"socket"来识别那些非命名套接字（如NAME列所见）。

第 11 章

shell脚本



如果你可以在shell中输入命令，那么你就可以使用shell脚本（或者叫Bourne shell脚本）。所谓shell脚本，就是将一系列命令写在一个文件当中，然后让shell从该文件读取命令，就像从终端读取一样。

11.1 shell 脚本基础

Bourne shell脚本一般以如下所示的行开头，它表示我们要用`/bin/sh`程序来执行脚本文件中的命令。（请确认脚本文件开头没有空格。）

```
#!/bin/sh
```


其中的`#!`叫作shebang，你在本书的其他脚本例子中也会看到这一行。你可以在该行之后列出任何你想让shell帮你执行的命令。例如：

```
#!/bin/sh
#
# Print something, then run ls

echo About to run the ls command.
ls
```

注解 若行以字符#开头，则表示该行为注释，即shell会忽略一行中#之后的所有东西。注释可用于解释脚本中难懂的部分。

创建好shell脚本并设置好它的权限之后，你就可以将该脚本放到你命令路径的某个目录中，

然后在命令行里输入脚本文件的名字来运行它。如果它在你当前目录下，你还可以用./script来运行它。当然也可以输入它完整的路径名来运行。

就像Unix系统的其他程序一样，你需要给shell脚本文件设置执行位和读位（以让shell能读取该文件）。最简单的设置方法是像下面这样：

```
$ chmod +rx script
```

如此执行chmod，就会使其他用户都能查看和执行script。如果你不希望这样，可转用绝对模式700（请参考2.17节来温习一下权限）。

基于这些基础知识，现在来看看shell脚本的局限性。

shell脚本的局限性

使用Bourne shell能方便地操控命令和文件。在2.14节中，我们看到了shell能重定向输出，这是shell脚本编程的重要元素之一。然而，shell脚本只是Unix编程的一个辅助工具，虽然它相当有用，但也有一些局限性。

shell脚本的主要优点是，它能使任务简单化、自动化，而不用你在命令行提示符下一条条地敲命令（这对批量处理文件很方便）。但如果你要分解字符串、做繁复的数学计算、做复杂的数据库交互，或者你想写函数以及复杂的控制结构，你最好还是用Python、Perl之类的脚本语言，或者awk，甚至C这样的编译型语言。（这是很重要的，所以本章将会多次提到这点）。

最后，注意你shell脚本的大小，尽量写得短小一些。Bourne shell脚本不应该写太大（虽然有人还是会写很大）。

11.2　引号与字面量

shell和shell脚本最令人困惑的一点就是，何时需要使用引号和其他标点符号，且为什么这么做。假设你想打印出字符串$100，于是你这么做：

```
$ echo $100
00
```

为什么会打出00呢？因为shell看到了$1，它是一个shell变量（这很快会讲到）。于是你可能会想给它加上双引号，让shell不管$1。但这样也不行：

```
$ echo "$100"
00
```

然后有人告诉你，要用单引号：

```
$ echo '$100'
$100
```

为什么这样就行呢?

11.2.1 字面量

引号通常用于创建字面量，即原封不动的字面义。就像上例中，用引号直接打出$，又例如你想把*传给grep之类的命令，或在命令中用分号（;）作为参数，都可以用单引号。

操作脚本或命令行时，先想想shell是如何执行一条命令的。

(1) 在执行之前，shell会查找其中的变量、通配符以及其他代词，如果有的话，就将它们进行替代。

(2) 将替换后的结果返回给命令。

涉及字面量的问题是很微妙的。假设你想查找/etc/passwd中符合正则表达式r.*t（意思是含有先r后t的行，例如含有用户名root、ruth或robot的行）的所有条目，你可以执行这样的命令:

```
$ grep r.*t /etc/passwd
```

大多数时候这样是行得通的，但有时却神秘地失效了。为什么呢？问题可能在于你当前的目录。如果当前目录包含名字如r.input和r.output的文件，那么shell就会将r.*t扩展为r.input和r.output，命令就会变成:

```
$ grep r.input r.output /etc/passwd
```

避免这种问题的关键是，找出可能被扩展的字符，然后用正确的引号来保护它。

11.2.2 单引号

创建字面量的最简单的方法就是用单引号将字符串包围，就像以下在grep命令中用*的字面量:


```
$ grep 'r.*t' /etc/passwd
```

对于shell来说，单引号之间的字符（包括空格），都会被当作一个单独的参数。所以，以下命令是行不通的，因为这样参数就只有一个了，它会让grep在标准输入中查找字符串r.*t /etc/passwd。

```
$ grep 'r.*t /etc/passwd'
```

当你想使用字面量时，请优先考虑单引号，它保证shell不会做任何替换，因此语法上也十分整洁。然而，可能有时你的需求会有点复杂，那么再考虑双引号。

11.2.3 双引号

双引号（"）跟单引号的效果差不多，只是shell会对双引号中的所有变量都进行扩展。你可

以试试以下命令，然后换成单引号再试试。

```
$ echo "There is no * in my path: $PATH"
```

以上命令中的$PATH会被转换，但*却没事。

注解　如果你使用双引号来打印大段文本，可考虑使用将在11.9节讲的here文档。

11.2.4　单引号的字面义

若想在Bourne shell中使用单引号的字面义，那就有点棘手了。有种解决方法是在单引号前加个反斜杠：

```
$ echo I don\'t like contractions inside shell scripts.
```

该反斜杠和单引号绝不能被任何单引号对包围，而'don\'t这样的语法也是错的。稀奇的是，双引号里可以用单引号，就像下面的例子（效果跟上面例子一样）：

```
$ echo "I don't like contractions inside shell scripts."
```

如果你想知道“不做转换”的一般法则，可参照以下做法：

(1) 将所有'（单引号）改成'\''（单引号、反斜杠、单引号、单引号）；

(2) 用单引号包围整个字符串。

因此，对于this isn't a forward slash: \这种麻烦的东西，可以这样做：

```
$ echo 'this isn'\''t a forward slash: \'
```

注解　反复提醒自己：引号中的任何东西都会被当成一个参数。所以，a b c是三个参数，而a "b c"则是两个。

11.3　特殊变量

大多数shell脚本都能够理解命令行的参数，以及懂得执行脚本里的命令。想要让脚本变成一个灵活的程序而不仅仅是一个命令列表，你需要知道特别变量的用法。这些特别变量跟2.8节介绍的变量很像，只是你不能直接改变它们的值。

注解　读完下面几节，你就会知道为什么shell脚本会有这么多特别的字符。如果你在脚本中遇到很难理解的语句，试试将其分拆。

11.3.1　单个参数：$1，$2，……

像$1、$2等所有以正整数命名的变量，都包含了脚本的参数值。例如，现有一脚本名叫pshow：

```
#!/bin/sh
echo First argument: $1
echo Third argument: $3
```

执行下列脚本，看它怎样打印参数：

```
$ ./pshow one two three
First argument: one
Third argument: three
```

shell的内置命令shift能删除第一个参数$1，并用后面的补上。说具体一点，就是$2变成$1，$3变成$2，如此类推。例如，现有一脚本名称叫shiftex：

```
#!/bin/sh
echo Argument: $1
shift
echo Argument: $1
shift
echo Argument: $1
```

运行来看看它是怎样运作的：

```
$ ./shiftex one two three
Argument: one
Argument: two
Argument: three
```


如你所见，shiftex每次只打印第一个参数，但因重复地使用shift，所以全部参数都打出来了。

11.3.2　参数的数量：$#

$#变量持有传给脚本的变量的数量，这对循环使用shift来遍历参数很有帮助。当$#为0时，就代表没有参数了，所以$1会是空。（参考11.6节与循环相关的知识。）

11.3.3　所有参数：$@

$@变量代表脚本接收的所有参数，可以整个传给脚本内的某个命令。例如，Ghostscript命令（gs）通常又长又复杂。假设你想以150dpi来栅格化一个PostScript文件，并用标准输出打印，同时又想为别人传其他选项预留一些可能性，你就可以这样写脚本，以允许额外的命令行选项：

```
#!/bin/sh
gs -q -dBATCH -dNOPAUSE -dSAFER -sOutputFile=- -sDEVICE=pnmraw $@
```

注解　如果脚本中有一行的长度超出了文本编辑器的范围，你可以用反斜杠（\）来分割它。例如，可将上面的例子改为：

```
#!/bin/sh
gs -q -dBATCH -dNOPAUSE -dSAFER \
   -sOutputFile=- -sDEVICE=pnmraw $@
```

11.3.4　脚本名：$0

$0持有脚本的名称，这对生成诊断信息很有帮助。例如，如果你希望脚本能报告$BADPARM变量出现非法值的情况，你就可以这样写，使得脚本名包含在报错信息中：

```
echo $0: bad option $BADPARM
```

所有的报错信息都应该转到标准错误。回忆2.14.1节讲的标准错误，其中提到的2>&1能让标准错误重定向到标准输出。若想反转这一过程，将上例的标准输出重定向到标准错误，可以使用1>&2：

```
echo $0: bad option $BADPARM 1>&2
```

11.3.5　进程号：$$

$$变量持有shell的进程号。

11.3.6　退出码：$?

$?变量持有shell执行的最后一条命令的退出码。它对掌控shell脚本来说非常重要，接下来就会讲到。

11.4　退出码

Unix程序退出时，会留一个*退出码*给启动该程序的父进程。它是一个数字，有时也叫*错误码*或*退出值*。当退出码是0时，通常表示程序运行正常。而如果是非正常结束，那么退出码通常会是非零数（也不总是这样，下面会说到）。

因为$?这一特殊变量含有最后一条命令的退出码，所以你可以通过shell提示符来查看它。

```
$ ls / > /dev/null
$ echo $?
0
$ ls /asdfasdf > /dev/null
ls: /asdfasdf: No such file or directory
$ echo $?
1
```

从上例可以看到,成功的命令返回0,失败的命令返回1(当然,我们假设你系统中没有/asdfasdf这样的目录)。

如果你要用某条命令的退出码,那就必须在执行完该命令后就马上用,不然就要先保存起来。例如,你连续运行了两次echo $?,那么第二条的结果肯定总是为0,因为第一个echo总是成功。

当需要让脚本非正常退出时,可使用exit 1,指定退出码1,并传给父进程。(你可以为不同情况最使用不同的退出码。)

要注意的是,有些程序,如diff和grep,正常退出时也会使用非零的退出码。例如,grep会在匹配时返回0,不匹配时返回1。对于这些程序来说,退出码1不代表出错;grep和diff使用2来代表出错。如果你不确定某个程序是否用非零退出码来指代运行成功,可参考其帮助手册,通常在EXIT VALUE或DIAGNOSTICS节中能找到解释。

11.5 条件判断

针对条件判断,Bourne shell有一种特别的代码结构,如if/then/ else和case语句。举个例子,以下带有if条件判断的脚本会检查第一个参数是否为hi:

```
#!/bin/sh
if [ $1 = hi ]; then
   echo 'The first argument was "hi"'
else
   echo -n 'The first argument was not "hi" -- '
   echo It was '"'$1'"'
fi
```

以上脚本的if、then、else和fi是shell关键字,其余都是命令。这样区分是很重要的,如[$1 = "hi"]就是一条命令,其中[字符是Unix系统中的一个实际存在的程序,而不是shell的语法。(其实这种说法不完全准确,很快你就会看到。但暂时先把它当作命令。)所有Unix系统都有一个命令叫[,它能进行条件测试。这个程序还有另外一个名字,叫test。对比[和test,会发现它们指向同一个inode,或者其中一个是另一个的符号链接。

11

理解了11.4节讲的退出码,才能看懂上例脚本的运作:

(1) shell执行if关键字之后的命令,获取了其退出码;

(2) 如果退出码是0,shell就会接着执行then关键字后的命令,直至遇到else或fi关键字;

(3) 如果退出码为非0,并且有else,就会执行else后的命令;

(4) 条件判断止于fi。

11.5.1 防范空参数

上例的条件判断还有个小问题,即$1可能会是空的,因为用户可能没有输入参数。没有参数的话,该脚本就会变成[= hi],令[出错并中止。你可用以下其中一种方法(两种都很常见)来修复:

```
if [ "$1" = hi ]; then
if [ x"$1" = x"hi" ]; then
```

11.5.2　使用其他命令来测试

if之后的内容必须是命令。因此，如果想将then放在同一行，必须在test命令后加上分号（;）。如果没有分号，test命令会将then当成是参数。（如果你不喜用分号，可将then放在下一行。）

除了[，还是有很多其他命令可用于测试的。以下就是一个用grep来测试的例子：

```
#!/bin/sh
if grep -q daemon /etc/passwd; then
    echo The daemon user is in the passwd file.
else
    echo There is a big problem. daemon is not in the passwd file.
fi
```

11.5.3　elif

if后面还可以用elif关键字，如下所示。但别串接太多的elif，因为你会发现后面11.5.6节的case更适合多条件分支的情况。

```
#!/bin/sh
if [ "$1" = "hi" ]; then
   echo 'The first argument was "hi"'
elif [ "$2" = "bye" ]; then
   echo 'The second argument was "bye"'
else
   echo -n 'The first argument was not "hi" and the second was not "bye"-- '
   echo They were '"'$1'"' and '"'$2'"'
fi
```

11.5.4　逻辑结构&&和||

你会经常看到这两种单行的条件判断结构：&&（“和”）和||（“或”）。&&结构是这样用的：

```
command1 && command2
```

这里，shell会执行command1，如果其退出码是0，就会接着执行command2。||结构也类似。如果||之前的命令返回了非0的退出码，||之后的命令就会被执行。

if测试中也常常用到&&和||。无论是用哪一个，都是由最后执行的命令的退出码来决定条件判断的走向。在&&的情况中，如果第一条命令失败了，shell就会用到if语句的退出码，但如果它成功了，则会用第二条命令的退出码。在||的情况中，如果第一条命令成功了，if就会用到它的

退出码，但如果它失败了，则会用第二条命令的退出码。

例如：

```
#!/bin/sh
if [ "$1" = hi ] || [ "$1" = bye ]; then
    echo 'The first argument was "'$1'"'
fi
```

如果你的条件判断包含了test（[）命令，如上例所示，那么你可以不用&&和||，而用-a和-o，这个下节会讲到。

11.5.5　测试条件

你已经见识过[的运作方式:测试成功返回0,测试失败返回1。你也知道了可以用[str1 = str2]来判断字符串是否相同。然而你还要知道，[这个有用的测试牵扯到文件的属性，使得shell脚本很适合处理文件。例如，以下代码就测试了一个文件是否为普通文件（非目录或特殊文件）:

```
[ -f file ]
```

你可能会看到有脚本将-f测试置于循环之中，像下面所示，用来测试当前目录里的所有项目（关于循环，后面很快就会讲到）:

```
for filename in *; do
    if [ -f $filename ]; then
        ls -l $filename
        file $filename
    else
        echo $filename is not a regular file.
    fi
done
```


你可在参数前加运算符!来测试相反的情况。例如，[! -f file]会在file不是普通文件的情况下返回true。此外，-a和-o对应“and”和“or”运算符（例如[-f file1 -a file2]）。

注解　因为test命令在脚本中应用广泛，所以很多Bourne shell版本（包括bash）都内置了test，这使得shell不需要为每个测试都单独执行一次命令，从而加快了速度。

test运算符有一大堆，它们可分为三类：文件测试、字符串测试和算术测试。info手册有完整的在线文档，但参考test(1)帮助手册更方便快捷。以下小节概述了主要的测试运算符。（已将不常用的忽略。）

文件测试

大多数文件运算符，像-f，被称为一元运算符，因为他们只要求一个参数：被测试的文件。例如下面这两个重要的运算符：

- ❏ -e：文件存在时返回true；
- ❏ -s：文件非空时返回true。

有些运算符能检查文件的类型，这意味着他们能看出一个东西是普通文件、目录，还是某种特殊设备（如表11-1）。另外，也有一些一元运算符是检查文件权限的，如表11-2。（关于文件权限，详见2.17节。）

表11-1　文件类型运算符

运算符	用于测试
-f	普通文件
-d	目录
-h	符号链接
-b	块设备
-c	字符设备
-p	命名管道
-S	套接字

注解　test命令会直达符号链接所指的文件（除非用-h测试）。意思是，如果link是某个普通文件的符号链接，则[-f link]返回true（退出码为0）。

表11-2　文件权限运算符

运算符	用于测试
-r	可读
-w	可写
-x	可执行
-u	Setuid
-g	Setgid
-k	“Sticky”

最后，还有三个二元运算符（需要两个文件作为参数）是用于文件测试的，但不常用。看看以下包含-nt（意思是“新于”）的命令：

```
[ file1 -nt file2 ]
```

如果file1的修改时间比file2要新，命令就返回true。而-ot（意思是“旧于”）则相反。想检测一个文件是否是另一文件的硬链接，可用-ef。

字符串测试

之前讲过，当两字符串相同时，二元字符串运算符=会返回true。而!=是在两字符串不相同时返回true。一元的字符串运算符有两个：

- ❏ -z：参数为空时返回true（[-z ""]返回0）；

- -n：参数为非空时返回true（[-n " "]返回1）。

算术测试

你需要认识到，等号（=）是用于检验字符串相等性的，而不是数字。因此，[1 = 1]返回0（true），但[01 = 1]返回false。当你想要检验两个数字是否相等时，请用-eq，而非等号：[01 -eq 1]会返回true。表11-3展示了所有比较数字的运算符。

表11-3 比较数字的运算符

运算符	当参数一与参数二相比，……时，返回true
-eq	相等
-ne	不等
-lt	更小
-gt	更大
-le	更小或相等
-ge	更大或相等

11.5.6 用case进行字符串匹配

关键字case用于另一种条件判断结构，在进行字符串匹配时，它格外好用。case条件判断不执行任何test命令，因此没有退出码。它是做模式匹配的，以下例子大概说明了它的情况。

```
#!/bin/sh
case $1 in
    bye)
        echo Fine, bye.
        ;;
    hi|hello)
        echo Nice to see you.
        ;;
    what*)
        echo Whatever.
        ;;
    *)
       echo 'Huh?'
       ;;
esac
```

其运行过程如下：

(1) 脚本将$1与每个)字符前的案例进行对比；

(2) 如果匹配，就会执行该案例后的命令，直至遇到;;时，跳到esac关键字；

(3) 整个条件判断结果以esac关键字结束。

案例可以是单个字符串（如上例的bye）或者是用|分隔的多个字符串（如果$1是hi或hello，则hi|hello返回true），你也可以使用*或?模式（如what*）。若想定义一个特定案例以外的默认案例，可用*，如上面代码的最后一个案例。

注解　每个案例都应以双分号（;;）结束，否则可能会导致语法错误。

11.6　循环

Bourne shell中有两种循环：for和while。

11.6.1　for循环

for循环（其实是“for each”循环）是最常见的，如下所示：

```
#!/bin/sh
for str in one two three four; do
    echo $str
done
```

以上的for、in、do和done都是shell关键字。它的运行过程是这样的：

(1) 将关键字in后的四个以空格区分的值赋值到变量str（使其为one）；

(2) 执行do和done之间的echo命令；

(3) 回到for那一行，把下一个值（two）赋予str，执行do和done之间的echo命令……这样重复着直至关键字in后的值都经历过。

此脚本的输出如下：

```
one
two
three
four
```

11.6.2　while循环

Bourne shell的while循环会使用到退出码，就像if一样。例如，以下脚本例子做了10次迭代。

```
#!/bin/sh
FILE=/tmp/whiletest.$$;
echo firstline > $FILE
while tail -10 $FILE | grep -q firstline; do
    # add lines to $FILE until tail -10 $FILE no longer prints "firstline"
    echo -n Number of lines in $FILE:' '
    wc -l $FILE | awk '{print $1}'
    echo newline >> $FILE
done

rm -f $FILE
```

以上代码会进行grep -q firstline的测试。只要该测试的退出码非零（本例中，若$FILE的最后10行不包含字符串firstline，则非零），循环就会结束。

你可以用break语句来打断while循环。Bourne shell还有一种类似while循环的until循环，它在退出码为零，而不是非零时结束循环。但是，你不应该经常使用while或until。实际上，你应该用awk或Python来代替while。

11.7　命令替换

Bourne shell能够做到将一个命令的标准输出重定向回该shell的命令行。意思是，你可以将一个命令的输出作为另一个命令的参数，或者用$()将命令包围，使该命令的输出能被当作变量使用。

以下例子就把一个命令的输出保存在了名为FLAGS的变量中。代码第二行的粗体部分展示了这种命令替换。

```
#!/bin/sh
FLAGS=$(grep ^flags /proc/cpuinfo | sed 's/.*://' | head -1)
echo Your processor supports:
for f in $FLAGS; do
    case $f in
        fpu)    MSG="floating point unit"
                ;;
        3dnow)  MSG="3DNOW graphics extensions"
                ;;
        mtrr)  MSG="memory type range register"
               ;;
        *)     MSG="unknown"
               ;;
    esac
    echo $f: $MSG
done
```

11

本例说明，你可以在命令替换中使用单引号和管道，看起来有点复杂。其中，grep命令的结果被发送到sed命令（详见11.10.3节）。该sed命令会将匹配.*:的内容都清空掉，然后得到的结果再发送给head命令。

注意不要过度使用命令替换。例如，请不要在脚本中用$(ls)，而是改用shell展开*，因为这样更快速。同样，如果你想将find命令查到的文件用作另一个命令的参数，请不要用命令替换，而考虑用管道将结果发到xargs，或者使用find的-exec选项（详见11.10.4节）。

注解　命令替换的传统做法是使用反引号（``）包围命令，你会在很多shell脚本中看到。$()语法是一种较新的形式，但它符合POSIX标准，所以更容易阅读和编写。

11.8　管理临时文件

有时我们需要用临时文件把输出暂存起来，以供后续的命令使用。请确保临时文件的文件名足够特别，以免被其他程序误认并误写入数据。

以下展示了如何使用mktemp命令来创建临时文件名。该脚本能显示出过往两秒内的设备中断信息：

```
#!/bin/sh
TMPFILE1=$(mktemp /tmp/im1.XXXXXX)
TMPFILE2=$(mktemp /tmp/im2.XXXXXX)

cat /proc/interrupts > $TMPFILE1
sleep 2
cat /proc/interrupts > $TMPFILE2
diff $TMPFILE1 $TMPFILE2
rm -f $TMPFILE1 $TMPFILE2
```

给予mktemp的参数是一个模板。mktemp会将XXXXXX转换成一个唯一的字符串，并以该名字创建出一个空文件。注意，该脚本将文件名保存在变量中，所以改文件名的话就只需要改一行而已。

注解　并非所有版本的类Unix系统都带有mktemp。如果移植代码后出现问题，最好先安装一下GNU coreutils包。

临时文件还有一个问题，就是如果脚本中止了，临时文件就可能不会被删掉。上例中，如果在第二个cat之前按下CTRL-C的话，就会使得临时文件留在/tmp里。请尽可能防止这种事情的发生。你可以用trap命令来创建一个信号处理器，在捕获到CTRL-C的信号时，删除临时文件，如下所示：

```
#!/bin/sh
TMPFILE1=$(mktemp /tmp/im1.XXXXXX)
TMPFILE2=$(mktemp /tmp/im2.XXXXXX)
trap "rm -f $TMPFILE1 $TMPFILE2; exit 1" INT
--snip--
```

你必须在该处理器中显式地使用exit来退出脚本，否则在trap过后，脚本仍会继续往下执行。

注解　mktemp不一定要有参数。无参数的时候，模板会以/tmp/tmp.前缀开头。

11.9　here 文档

假设你想打印大段文字，或将大段文字传给另一个命令，与其用一大堆echo命令，不如用shell的here文档。如下例所示：

```
#!/bin/sh
DATE=$(date)
cat <<EOF
Date: $DATE

The output above is from the Unix date command.
It's not a very interesting command.
EOF
```

上例的粗体部分（即<<EOF）就控制着here文档。shell会把<<EOF之后的标准输入都重定向到<<EOF之前的命令（本例中的cat）。重定向截止于EOF再次单独出现之时。你也可以指定EOF之外的标志，但要记住，起始标志与结束标志必须一样。此外，一般约定标志是用大写的。

注意here文档中的$DATE。here文档中的shell变量都会被展开，这在打印包含大堆变量的报告时十分有用。

11.10　重要的 shell 脚本工具

有一些程序在shell脚本中相当有用。像basename之类的工具，是很少单独使用的，所以你可能只会在脚本中看到它。而awk之类的工具，则是在命令行上也常用到。

11.10.1　basename

想要去掉文件的扩展名，或者去掉路径全名中的目录部分，可以使用basename。试试以下命令，看basaname能输出什么结果：

```
$ basename example.html .html
$ basename /usr/local/bin/example
```


以上两条命令都会返回example。其中第一条会将example.html中的后缀.html去掉，第二条会将全路径中的目录部分去掉。

以下例子展示了如何在脚本中使用basename把GIF图像转换成PNG格式。

```
#!/bin/sh
for file in *.gif; do
    # exit if there are no files
    if [ ! -f $file ]; then
        exit
    fi
    b=$(basename $file .gif)
    echo Converting $b.gif to $b.png...
    giftopnm $b.gif | pnmtopng > $b.png
done
```

11.10.2　awk

awk的功能并不单一，实际上它是一种很强大的编程语言。不幸的是，它就像一门失落的艺术，正被更大型的语言（如Python）所取代。

关于awk的书有很多，如Alfred V. Aho、Brian W. Kernighan和Peter J. Weinberger的*The AWK Programming Language*（Addison-Wesley，1988）。然而，很多人都只是用awk来做一件事——从输入流中截取单一的字段，就像这样：

```
$ ls -l | awk '{print $5}'
```

这句命令会将ls的输出的第五个字段（文件大小）打印出来，其结果就是一列文件大小。

11.10.3　sed

sed（意思是流编辑器）程序是一个自动的文本编辑器，它能根据一些表达式来修改输入流（文件或标准输入）的内容，并将修改结果打印到标准输出。sed有很多方面都很像Unix原始的文本编辑器ed。它有大量的运算符、匹配工具和定位功能。跟awk一样，讲sed的书也有很多，而 Arnold Robbins的*sed & awk Pocket Reference*, 2nd edition（O'Reilly，2002）就是一本涵盖sed和awk的快速指南。

虽然sed是个很大的程序，本书不能对其深入分析，但讲解它的运作是很简单的。我们把地址和操作组合起来，当作sed的一个参数。其中，地址是行的集合，而命令则代表了我们对这些行所要执行的操作。

sed的常见用法，是根据一个正则表达式（详见2.5.1节）进行内容替换，像下面这样：

```
$ sed 's/exp/text/'
```

如果你想将/etc/passwd的第一个冒号替换成%，可以这么做：

```
$ sed 's/:/%/' /etc/passwd
```

如果你想将/etc/passwd的所有冒号都替换成%，可以在末尾加上g修饰符：

```
$ sed 's/:/%/g' /etc/passwd
```

以下是按行处理的例子，它读取/etc/passwd的内容，并把三到六行去掉之后，打印到标准输出：

```
$ sed 3,6d /etc/passwd
```

本例中，3,6是地址（行的范围），而d是运算符（delete）。如果没有写地址的话，sed就会在所有行上应用该操作。最常用的sed运算符应该是s（查找与替换）和d了。

你也可以用正则表达式作为地址。以下命令会剔除所有匹配正则表达式exp的行：

```
$ sed '/exp/d'
```

11.10.4 xargs

当你把海量的文件当作一个命令的参数时，该命令或者shell可能会告诉你缓冲不足以容纳这些参数。解决这个问题，可用xargs，它能对自身输入流的每个文件名逐个地执行命令。

很多人会把xargs和find一起用。例如，以下脚本能帮你验证当前目录树中所有以.gif结尾的文件是否真的是GIF图像：

```
$ find . -name '*.gif' -print | xargs file
```

上例，xargs会执行file命令。不过，这样执行的话可能会出错，或造成安全问题，因为文件名可能包含空格或换行符。所以，请改用以下形式。这样，find的输出和xargs的参数的分界符就会是空字符，而不是换行符了。

```
$ find . -name '*.gif' -print0 | xargs -0 file
```

xargs会发起多个进程，所以如果是处理大量文件的话，不要期望它性能有多好。

你可能需要在xargs的末尾加上两个连字符（--），以防有文件名以连字符开头。程序会将双连字符后的参数都当作文件名，而不是选项。但是注意，不是所有的程序都支持双连字符。

使用find的时候，可以用选项-exec，而不用xargs。不过它的语法有点麻烦，你需要加上一个{}来代表文件名，以及一个字面义的;来表示命令的结束。以下是用find来改写上例：

```
$ find . -name '*.gif' -exec file {} \;
```

11.10.5 expr

如果需要在shell脚本中进行算术操作，可使用expr（它甚至能进行字符串操作）。例如，命令expr 1 + 2会输出3。（执行expr --help，查看所有的运算符。）

其实expr做算术操作效率很低。如果你经常要做算术操作，你可以改用Python，而不用shell脚本。

11.10.6 exec

exec命令是shell内置的，它会用其后的程序的进程来取代你当前的shell进程。它用到了第1章讲到的exec()。此功能的目的是节省系统资源，但记住，它是没有返回值的。当你在shell脚本中运行exec的时候，该脚本以及运行该脚本的shell都会失踪，而被exec后的命令顶替。

你可以在shell窗口中运行exec cat来看看它的效果。当你按下CTRL-D或CTRL-C的时候，shell窗口就会消失，因为已经没有任何子进程了。

11.11　子 shell

假设你要稍微改动一下环境变量，但又不想做永久的改动，你可以用shell变量来暂存和恢复部分环境变量（例如path或当前目录）。但这种做法很笨拙。简单的做法是使用子shell，即新开一个进程来运行一两条命令。新shell复制了源shell的环境变量，而在其退出时，你对其环境变量的任何更改却不会影响到源shell。

将命令置于括号中，即可运行子shell。例如下面这行命令，它在uglydir目录里运行了uglyprogram，而并不影响源shell：

```
$ (cd uglydir; uglyprogram)
```

以下是一个为避免永久改动path而用子shell来操作的例子：

```
$ (PATH=/usr/confusing:$PATH; uglyprogram)
```

用子shell来暂时改变一个环境变量是很常见的，且因此还诞生了一种避免写子shell的内置语法：

```
$ PATH=/usr/confusing:$PATH uglyprogram
```

管道和后台进程可以与子shell一同使用。下面例子的意思是，用tar将orig下的整个目录树打包，并在target目录中解包。这一做法很高效地复制了orig中的文件和目录（这种做法很有用，因为它保留了权限，而且一般会比cp -r快）。

```
$ tar cf - orig | (cd target; tar xvf -)
```

警告　运行此命令之前，请仔细检查，确保目标命令存在，并且与源目录是完全分开的。

11.12　在脚本中包含其他文件

点（.）运算符可以在脚本中包含其他文件。例如，下例表明运行了config.sh里的命令：

```
. config.sh
```

这种“包含”语法不会开启子shell，而且便于在脚本中调用配置文件。

11.13 读取用户输入

read命令可以将标准输入的内容读取到变量中。例如，以下命令会将输入置于$var：

```
$ read var
```

将此内置命令与本书未提及的其他shell特性一起使用，能实现很有用的功能。

11.14 什么时候（不）应该使用 shell 脚本

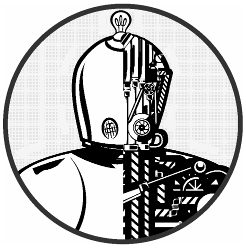
shell的功能之丰富，使得我们难以用区区一章讲完它的所有重点。如果你想知道shell还能做些什么，不妨参考一些关于shell编程的书籍如Stephen G. Kochan和Patrick Wood的*Unix Shell Programming*, 3rd edition（SAMS Publishing，2003），或者Bran W. Kernighan和Rob Pike的*The UNIX Programming Environment*（Prentice Hall，1984）中的shell脚本部分。

然而，在某些情况下（尤其是你准备使用read时），你应该反问一下自己，你是否在使用正确的工具来完成这项工作。请记住shell脚本的强项：操控简单的文件和命令。如前面提到的，当你发现你的脚本写得有点繁琐，特别是涉及复杂的字符串或数学处理时，或许你就该试试Python、Per或awk之类的脚本语言了。

11

第 12 章

在网络上传输文件



本章考察在网络上的机器之间传输和分享文件的各种方法。我们会先介绍一些scp和sftp以外的文件复制工具，接着再看看真正的文件分享，即机器之间如何分享目录。

这里之所以要介绍传输文件的不同方法，是因为它们针对的是不同的问题。有时你只想向不甚了解的机器提供一种快速、暂时的访问途径，有时你需要高效地管理大量目录结构，或者有时你需要的是固定的访问。

12.1　快速复制

假设你想在你的网络上从一台机器复制文件到另一台机器，并且不打算复制回来或进行其他操作——你要的只是快，那么，使用Python就是一种方便的做法。直接打开该文件的目录，然后执行：

```
$ python -m SimpleHTTPServer
```

这会启动一个基本的网页服务器，使得网络上的浏览器能够看到该目录。它通常使用8000端口，所以，如果你在10.1.2.4的机器上运行，那么，到http://10.1.2.4:8000就可以获取你想要的东西。

12.2　rsync

如果你想移动整个目录，可用scp -r。如果想提高速度，那就用tar和管道：

```
$ tar cBvf - directory | ssh remote_host tar xBvpf -
```

这种做法可行，但不灵活。具体来说，传输结束时，在远端的那个目录可能并不真的就是本目录的副本。如果远端已经有了上例提到的directory目录，并且里面包含其他文件，那么传输完成时，那些文件会是依然存在的。

如果你日常需要做这种事情（并希望自动化），可用一些专业的同步系统。rsync是Linux平台上标准的同步工具，它功能多样，性能强大。接下来我们会介绍一些实用的rsync操作模式和它的一些奇特之处。

12.2.1 rsync基础

要使rsync在两台主机之间运作，必须让它在源机器和目标机器上面都安装。接着，你还需要知道从一台机器访问另一台机器的方法。而最简单的方法，就是使用远端的shell账号（假设你想用SSH方式传输）。其实，rsync还可以在同一台机器上进行文件和目录的复制，例如从一个文件系统复制到另一个文件系统。

表面上，rsync命令跟scp没什么区别。事实上，rsync可以使用相同的参数。例如，想复制一组文件到你名为host的机器的home目录，就可以输入：

```
$ rsync file1 file2 ... host:
```

在所有现代操作系统中，rsync都假设你是用SSH连接远端的。

小心这种报错：

```
rsync not found
rsync: connection unexpectedly closed (0 bytes read so far)
rsync error: error in rsync protocol data stream (code 12) at io.c(165)
```

这个信息是说，在远端找不到rsync。如果远端安装了rsync，但没在路径中，用--rsync-path=path来指定远端中它的位置。

如果你的远端账号跟本机的不一样，那就在主机名前加上user@，这里说的user指的是你的名为host的远端账号：

```
$ rsync file1 file2 ... user@host:
```

除非你加了其他选项，不然rsync就只复制文件。事实上，如果你用上我们讲过的那些选项，并加上一个叫dir的目录为参数，你就会收到以下信息：

```
skipping directory dir
```

想要完整地、递归地传输整个目录——包括符号链接、权限、模式、设备——那就用-a选项。还有，如果你想复制到其他地方，而不是你的home目录，你可以在host后面加上具体地址，就像这样：

12

```
$ rsync -a dir host:destination_dir
```

复制目录是需要技巧的，如果你不清楚传输过程中会发生什么，你可以使用-nv选项组合。-n选项会让rsync进入“空跑”模式，即模拟复制而非真的复制。-v选项则是冗长模式，它会将涉及的文件和传输过程的细节显示出来：

```
$ rsync -nva dir host:destination_dir
```

输出大概会是这样：

```
building file list ... done
ml/nftrans/nftrans.html
[more files]
wrote 2183 bytes read 24 bytes 401.27 bytes/sec
```

12.2.2　准确复制目录结构

默认情况下，rsync复制文件或目录时是不关心目标目录的原本内容的。例如，如果你把一个包含文件a和b的目录d复制到另一台机器，而该机器已有了一个目录d，并且其中包含文件c，那么经rsync传输过后，目标目录就会包含有a、b、c三个文件。

所谓准确复制，就是将目标目录比源目录多出的内容清除掉，例如上例的c文件。要想实现这种效果，你可以使用--delete选项：

```
$ rsync -a --delete dir host:destination_dir
```

警告　这样做是有点危险的，你需要检查一下会不会误删文件。记住，如果你不确定传输是否能达到理想结果，可先以-n选项做一下测试，看看是否会删除文件。

12.2.3　以斜杠结尾

在rsync中把目录作为源时，要特别小心。想想我们谈过的传输目录的基本做法：

```
$ rsync -a dir host:dest_dir
```

执行过后，host机器的dest_dir下就会有了目录dir。图12-1就展示了这种情况（假设目录里面有文件a和b）。而如果你的源目录是以斜杠（/）结尾的话，则是另一番景象：

```
$ rsync -a dir/ host:dest_dir
```

这样的话，rsync就会直接将dir里的内容全部复制到dest_dir中，而不会先在dest_dir建立dir目录。因此，你可把它看成是在本地文件系统中执行cp dir/* dest_dir。

假设你有一个包含文件a和b的目录dir，然后你执行了以斜杠结尾的那个版本的命令：

```
$ rsync -a dir/ host:dest_dir
```

传输过后，dest_dir里就会有a和b的副本，但并无目录dir。而如果你没有加斜杠，那么dest_dir得到的将会是目录dir的副本（里面包含a和b），即远端生成了dest_dir/dir/a和dest_dir/dir/b这样的路径和文件。从图12-2我们可以看出，加上斜杠的结果是不同于图12-1的。

如果传输目录和文件时不小心加上了斜杠，是没什么大不了的，只是可能有点麻烦而已。你需要到远端去建立dir目录，并将文件放回去。需要担心的是，如果你不幸将它跟--delete选项结合使用，那就可能会误删无辜的文件。

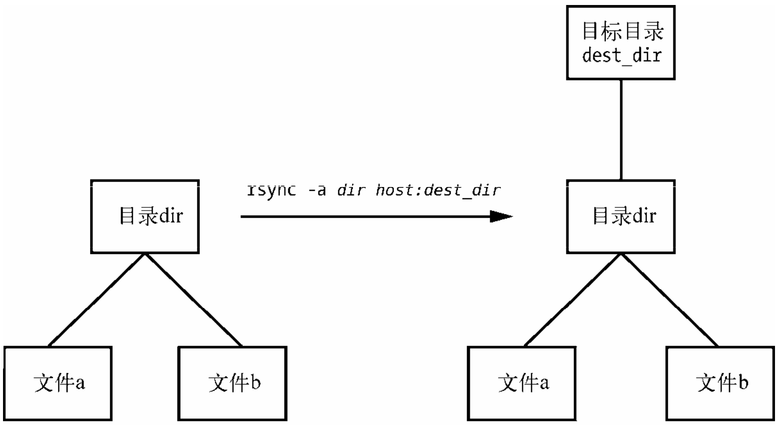


图12-1　一般的rsync复制

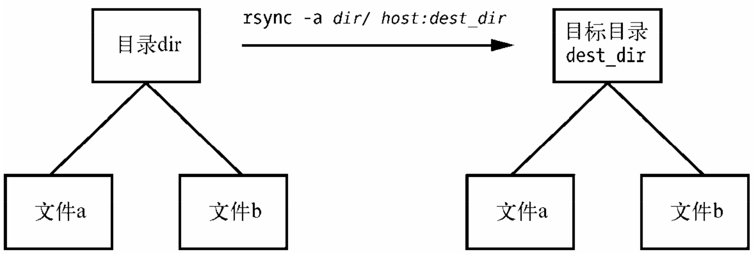


图12-2　斜杠结尾的效果

注解　注意shell中的文件名补全功能。GNU readline以及其他补全库可能会给完整的目录名加上斜杠。

12.2.4　排除文件与目录

rsync有个很重要的功能，就是它可以排除某些文件与目录的传输。比如说，你想将一个本地目录src传输到host，但想排除其中叫.git的任何东西。那你可以这样做：

```
$ rsync -a --exclude=.git src host:
```

注意，这条命令会排除所有名为.git的文件与目录，因为这里--exclude获得的是模式，而非一个绝对的文件名。要想排除一个特定的项目，那就以“/”开头来指定一个绝对路径，就像这样：

```
$ rsync -a --exclude=/src/.git src host:
```

注解　/src/.git开头的/不是指你系统的home目录，而是指源目录所在的位置。

以下还有一些按模式进行排除的技巧。

- --exclude选项可以有多个。
- 常用的模式可保存在文本文件中（一行一个模式），并用--exclude-from=file。
- 只想排除名为item的目录，不想排除名为item的文件，可以斜杠结尾：--exclude=item/。
- 模式匹配是以一个完整的文件名为单位的，模式可包含通配符。例如，t*s能与this匹配，但不与ethers匹配。
- 如果你发现你要排除的内容太多了，你也可以用--include来指定要包含的文件或目录。

12.2.5　合并、检查及冗长模式

为了提高效率，rsync会先快速检查一下目标位置是否已包含源内容。这种快速检查只看文件大小及最后修改时间。在你第一次将整个目录结构传输到远端时，rsync发现远端并不存在任何这些文件，于是它就整份传输过去。你可以用rsync -n来验证。

传输过一次之后，再用rsync -v做一次。你会发现这次的文件列表为空，那是因为目标位置已有了源内容，并且修改时间也一致。

当源文件与目标文件不一致时，rsync会对远端的文件进行覆写。那么，之前提到的快速检查功能就可能不够用了，因为你可能还需要更准确地验证两边文件是否一样，以免rsync错误地略过它们，又或者说你想要更多安全保障。这时，你可以求助于以下这些选项。

- --checksum（缩写-c）：通过计算文件的校验和（大部分情况下是唯一的）来检查一致性。这会消耗更多的I/O和CPU资源，但如果你想准确地传输，又担心文件大小不足以判断一致性，那么这个选项就是必须的。
- --ignore-existing：不覆写已存在的文件。
- --backup（缩写-b）：不覆写已存在的文件，只在传输前给它们加上~后缀，为它们重命名。

- --suffix=s：将--backup用的后缀~改为s。
- --update（缩写-u）：当目标文件的修改时间比源文件的更早时，才进行覆写。

在没有使用什么特别选项时，rsync的运行会是悄无声息的，只在遇到错误时才会有输出。你可以用rsync -v（冗长模式）来显示传输过程的细节，甚至用rsync -vv来显示得更细。（v越多则越详细，但两个v通常已足够。）想在传输过后得到一些综合信息，可用rsync --stats。

12.2.6 压缩

很多人喜欢在用-a的时候加上-z，以在传输前先进行压缩：

```
$ rsync -az dir host:destination_dir
```

某些情况下，压缩是能加快传输速度的，例如说上行缓慢，或者说延迟很多，而你又要传输大量文件的情况时。然而，如果网络很快，那么压缩和解压的过程会占用较多CPU时间，此时反而不做压缩会传得更快些。

12.2.7 限制带宽

上传大量数据时，可能会堵塞上行线路。尽管这并不占用下行线路，但如果不限制上传速度，其实也是会影响下行的。因为发出的TCP包（如HTTP请求）需要与你的文件传输争夺带宽。要想避免这种情况，可用--bwlimit来给你的上行线路留点空间。例如，若要令其上行带宽的上限为10 000Kbps，可这么做：

```
$ rsync --bwlimit=10000 -a dir host:destination_dir
```

12.2.8 传文件到你的计算机

rsync不仅能从本机传文件到远端，还能从远端传文件到本机，你只需将远端主机名和远端文件路径作为第一个参数即可。因此，想将远端host的src_dir传到本机的dest_dir，就可以这样写：

```
$ rsync -a host:src_dir dest_dir
```

注解 之前也提过，不填主机名host:的话，意味着是在本机上进行文件复制。

12.2.9 更多有关rsync的话题

无论何时，rsync都应该是文件复制的首选工具之一。rsync的批量模式很有用，它会用到一些辅助文件来记录状态日志。状态日志能使中断的长时传输恢复得更快捷。

`rsync`还能用于备份。例如，你可以接上一些网络硬盘，如Amazon的S3，到你的Linux系统，然后使用`rsync --delete`来定期地进行文件同步，这样的备份系统是很高效的。

没谈到选项还有很多。想大概了解下的话，可运行`rsync --help`来查看。此外，rsync(1)帮助手册及其主页http://rsync.samba.org/也提供了详细的参考信息。

12.3　文件共享

你的网络中应该不只有你一台Linux机器，而在拥有多台机器的网络中，很多时候我们都需要进行文件共享。在本章剩下的内容中，我们主要关注与Windows或Mac OS X的文件共享，因为了解Linux与其他平台的适配是挺有趣的。而对于在Linux机器之间分享文件，或访问网络区域存储（Network Area Storage，以下简称NAS）中的文件，我们会简单介绍一下，看一下怎样以客户身份使用网络文件系统（Network File System，以下简称NFS）。

12.4　用 Samba 分享文件

如果你有机器运行着Windows，你可能想允许它通过标准的Windows网络协议——服务器消息块（Server Message Block，以下简称SMB）来访问你Linux系统的文件和打印机。Mac OS X也支持SMB文件共享。

Unix的标准文件共享套件叫作Samba。它不只能让Windows机器访问Linux系统，还能反过来——用Linux上的Samba客户端访问和打印Windows服务器上的文件。

要建立Samba服务器，可跟从以下步骤：

(1) 创建smb.conf文件；

(2) 在smb.conf中加入文件共享小节；

(3) 在smb.conf中加入打印机共享小节；

(4) 启动Samba守护进程nmbd和smbd。

当你安装发行版的Samba包时，你的系统会执行以上的步骤，并给予Samba服务器合理的设置。然而，它无法为你决定你想要共享什么东西。

注解　本章关于Samba的介绍是很简单的，而且仅限于让单个子网中的Windows机器通过网上邻居来查看Linux机器。因为访问控制和网络拓扑也有各种可能性，所以配置Samba的方法无穷无尽。想知道大规模服务器的配置方法，参考*Using Samba*, 3rd edition（O'Reilly，2007）这本书，里面谈到的东西很广泛。另外，也可参考Samba的网站http://www.samba.org/。

12.4.1　配置服务器

Samba的核心配置文件是smb.conf，大多数发行版都会将它置于etc目录中，如/etc/samba。然而，也有可能会被放在了lib目录中，如/usr/local/samba/lib。

smb.conf的格式类似于XDG风格（如systemd配置文件的格式），并分为不同的小节，每节的名称以方括号包围（例如[global]和[printers]）。[global]节包含了应用于整个服务器和所有共享的通用选项。这些选项主要是关于网络配置和访问控制的。以下[global]例子展示了如何设置服务器名字、描述和工作组：

```
[global]
# server name
netbios name = name
# server description
server string = My server via Samba
# workgroup
workgroup = MYNETWORK
```

这些参数的含义如下所示。

- netbios name：服务器名字。如果没填，则Samba会用Unix主机名。
- server string：关于服务器的简短描述。默认是Samba的版本号。
- workgroup：SMB工作组名字。如果你在Windows域中，那就把该参数设为Windows域的名字。

12.4.2 服务器访问控制

你可以在smb.conf中添加选项来限制那些能访问你Samba服务器的机器和用户。以下列举的是一些包括了很多可设在[global]节和访问控制节的选项（稍后讲解）。

- interfaces：使Samba监听某个网络或接口。例如：

```
interfaces = 10.23.2.0/255.255.255.0
interfaces = eth0
```

- bind interfaces only：使用interfaces选项时，将此选项设为yes，以使Samba只对指定的接口服务。
- valid users：只允许指定的用户访问Samba，例如：

```
valid users = jruser, bill
```

- guest ok：设为true则会使共享对网络上的匿名用户可见。
- guest only：设为true则是只允许匿名用户访问。
- browseable：设置网络浏览器是否可见共享资源。设为no时，要指定具体名字才能访问共享内容。

12.4.3 密码

一般来说，访问Samba服务器应该先通过密码验证。不幸的是，Unix的基本密码系统跟

Windows的是不一样的，所以，除非你指定了明文的网络密码或Windows服务器验证密码，否则你必须要有另一套密码系统。本节会教你如何用Samba的Trivial Database（即TDB）后端建立一套密码系统，它很适合于小型网络。

首先，在smb.conf的[global]节中创建以下条目，来定义Samba密码数据库的特性：

```
# use the tdb  for Samba to enable encrypted passwords
security = user
passdb backend = tdbsam
obey pam restrictions = yes
smb passwd file = /etc/samba/passwd_smb
```

这几行代码能让你通过smbpasswd来操作Samba密码数据库。obey pam restrictions参数保证了用户通过smbpasswd修改密码时需要遵照PAM的规则。而对于passdb backend参数，你还可以在冒号后指定TDB文件的路径，如tdbsam:/etc/samba/private/passwd.tdb。

注解　*如果你使用Windows域，你可以设置security = domain以令Samba使用域的用户名，而无需密码数据库。然而，这要求两端的用户名必须要一致。*

添加和删除用户

为了让Windows用户能访问Samba服务器，你必须先用smbpasswd -a来把windows用户的用户名添加到数据库中：

```
# smbpasswd -a username
```

username参数必须是Linux系统中有效的用户名。

跟passwd程序一样，smbpasswd也会让你为新用户输入两次密码。如果密码能通过必要的安全检查，则新用户就会被创建。

要移除用户，则用-x选项：

```
# smbpasswd -x username
```

-d选项可令用户暂时失效；而-e选项则可将用户重新激活：

```
# smbpasswd -d username
# smbpasswd -e username
```

修改密码

超级用户可不加任何选项或关键字就直接更改任何一个用户的Samba密码：

```
# smbpasswd username
```

然而，如果Samba服务器是在运行中，则任何用户都可通过在命令行输入smbpasswd来修改自

己的密码。

最后，配置中还有一点是需要注意的，即如果你在smb.conf中看到这样的一行，那就要小心了：

```
unix password sync = yes
```

这行会使smbpasswd在改Samba密码的同时也改掉Linux密码。这可能会给人带来困扰，尤其是当用户将密码修改为非Linux登录密码后，却发现不能登录Linux了。有些发行版的Samba包正是默认设为yes的！

12.4.4 启动服务器

如果你不是用发行版的Samba包来安装，那么你可能需要自己启动服务器。你要做的是用以下参数来运行nmbd和smbd，而其中的smb_config_file是你smb.conf文件的完整路径：

```
# nmbd -D -s smb_config_file
# smbd -D -s smb_config_file
```

nmbd守护进程是一个NetBIOS名字服务器，而smbd才是真正处理共享请求的进程。-D选项用于指定它们以守护进程模式运行。如果你在smbd运行期间修改了smb.conf文件，你可以用HUP信号通知守护进程重读配置，或用你发行版的服务重启命令（例如systemctl或initctl）来做到。

12.4.5 诊断和日志文件

如果Samba服务器启动过程中出错，那么错误信息就会出现在命令行上。但运行时的诊断信息是去到log.nmbd和log.smbd文件的，它们通常在/var/log目录中，如/var/log/samba。在那里你还可以看到其他日志文件，例如每个客户单独的日志文件。

12.4.6 配置文件共享

12

要向SMB客户输出一个目录（即与客户共享一个目录），需要在你smb.conf文件中添加如下这样的一个小节。其中label是该共享的名称，path是目录的完整路径。

```
[label]
path = path
comment = share description
guest ok = no
writable = yes
printable = no
```

以下参数对于目录共享是很有帮助的。

- guest ok：允许访客访问，与public参数同义。
- writable：设为yes或true，表示该共享可读可写。千万不要给一个访客可访问的共享设置可读可写。

- printable：设置为共享打印机。如果是目录共享，那就要设为no或false。
- veto files：符合模式的文件将不参与共享。每个模式用斜杠包围（像/pattern/这样）。例如，veto files = /*.o/bin/这条命令会屏蔽对象文件以及所有叫bin的文件和目录。

12.4.7　home目录

你可以在smb.conf文件里添加[homes]节，以使home目录能被共享。这个节大概是这样：

```
[homes]
comment = home directories
browseable = no
writable = yes
```

默认地，Samba会读取登录用户的/etc/passwd条目来获知他们的home目录。然而，如果你不想用默认的做法（即你想让Windows的home目录与Linux的不同），那你可以在path参数处使用%S。例如，下例就可以将一个用户的[homes]目录切换成/u/user：

```
path = /u/%S
```

Samba会将%S替换成当前的用户名。

12.4.8　共享打印机

你可以通过在smb.conf文件中加入[printers]节来共享打印机。以下是使用标准Unix打印系统CUPS时[printers]节的配置：

```
[printers]
comment = Printers
browseable = yes
printing = CUPS
path = cups
printable = yes
writable = no
```

要使用printing = CUPS参数，你的Samba必须配置和连接了CUPS库。

注解　根据你的需要，你可能还想配置guest ok = yes，以允许访客访问你的打印机，而不是直接派发账号密码，因为用防火墙限制打印机访问是很容易的。

12.4.9　使用Samba客户端

Samba的客户端程序smbclient可以访问Windows共享。当你需要与Windows服务器交互时，这个工具能让你仍按Unix风格来工作。

用-L选项列出名为SERVER的远端的共享：

```
$ smbclient -L -U username SERVER
```

如果你的Linux用户名与远端的用户名一样，就不需要-U username。

输入命令后，smbclient就会问你密码。若要以访客身份访问，就直接回车，否则输入SERVER的密码。成功后，你就会看到以下结果：

```
Sharename   Type      Comment
---------   ----      -------
Software    Disk      Software distribution
Scratch     Disk      Scratch space
IPC$        IPC       IPC Service
ADMIN$      IPC       IPC Service
Printer1    Printer   Printer in room 231A
Printer2    Printer   Printer in basement
```

可以根据Type字段来搞清楚每行的作用，并只关注Disk和Printer共享（IPC共享属于远程管理）。这个列表包含了两个共享硬盘和两个共享打印机。可用Sharename一列的名字来访问它们。

12.4.10　作为客户去访问文件

如果你只是随意访问共享硬盘上的文件，可用以下命令。（同样，如果你Linux用户名与远端的一致，可以去掉-U username。）

```
$ smbclient -U username '\\SERVER\sharename'
```

成功执行后，你会看到以下提示，意味着你可以开始传输文件了：

```
smb: \>
```

在此种传输模式下，smbclient跟Unix的ftp命令是很像的，你可以运行下列命令。

- get file：将远端的文件file复制到本机当前目录。
- put file：将本机文件file复制到远端。
- cd dir：跳到远端的dir目录。
- lcd localdir：跳到本机的localdir目录。
- pwd：打印出远端的当前目录，包括服务器与共享的名称。
- !command：在本机执行命令command。可以用!pwd和!ls来查看本机的当前目录和目录中的文件。
- help：显示所有可用命令。

使用CIFS文件系统

如果你要经常访问Windows服务器上的文件，你可以通过mount来将一个共享附加到你的系统上，语法如下。注意是SERVER:sharename格式，而不是一般所见的\\SERVER\sharename。

```
# mount -t cifs SERVER:sharename mountpoint -o user=username,pass=password
```

12

要像这样使用mount，你的Samba要先能使用通用互联网文件系统（Common Internet File System，以下简称CIFS）。大多数发行版会在Samba包以外提供。

12.5 NFS 客户端

Unix系统中标准的文件共享系统是NFS。不同版本的NFS适用于不同的情境。你可以通过TCP和UDP来提供NFS服务，并加上一大堆验证和加密技术。正因为选择繁多，所以NFS是个很大的话题，所以我们只讲讲NFS客户端的一些基本知识。

想在一个服务器上通过NFS挂载一个远程目录，可使用与挂载CIFS相同的语法：

```
# mount -t nfs server:directory mountpoint
```

技术上来说，你应该不用指定-t nfs，因为mount会帮你识别。但你可能也想了解帮助手册nfs(5)里的选项。（你会看到sec选项可使用好几种不同的安全策略。很多小型的、封闭的网络的管理员会使用基于主机的访问控制。而更复杂的那些方法，如基于Kerberos的验证，需要额外的配置文件。）

当你发现网络上的文件系统使用量很大时，那就建立自动挂载吧，使得文件系统在被需要使用时才挂载，以避免开机时出现堵塞而产生依赖问题。传统的自动挂载工具是automount，它的新版叫作amd，但他们都正在被systemd的自动挂载单元取代。

12.6 有关网络文件服务的选择与局限的更多内容

创建NFS服务器来共享文件比使用NFS客户端要更复杂。你需要运行服务器守护进程（mountd和nfsd），并建立/etc/exports文件来反映你分享的目录。然而，我们不会在此话题上展开太多，因为直接购买一个NAS设备会更方便。这种设备很多都是基于Linux的，所以它们自然支持NFS服务器。供应商会附加一些他们自制的管理工具，来帮你轻松解决一些无聊的任务，如RAID配置和云备份。

讲到云备份，有种网络文件服务叫云存储。如果你需要自动备份的功能而不要求什么高性能，那它就很适合你。尤其是那些你不常用的文件，通通可以放到云存储中。你可以像使用NFS那样使用它。

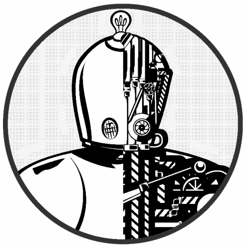
NFS和其他文件共享系统应付一般的使用是没问题的，但别指望它们拥有强大的性能。对大文件的只读访问通常没什么问题，例如流式地读取音频和视频，因为那些内容是连续的、可预测的，并且不需要太多的服务器与客户端之间的往返。只要网络够快，且客户端的内存够大，服务器就能持续地供应你想要的数据。

使用本地存储能更快速地处理涉及大量小文件的任务，例如编译软件包和启动桌面环境。当网络变得更大、机器间的交流变得更多时，事情就变得更复杂，因为你需要在便捷程度、性能高低及管理难易之间作出权衡。

第 13 章

用户环境

13



本书主要关注Linux系统的服务器进程和交互式的用户会话。但系统与用户最终是需要有交汇点的。启动文件在这一点上就发挥着重要的作用，因为它们为shell和其他交互式程序设定了默认的参数。它们决定了用户登录时所能看到的系统的模样。

大多数用户平时是不太关心启动文件的，只会在想要添加一些方便功能时才接触它们，例如添加别名。随着时间推移，这些文件便充斥了不必要的变量和测试，从而导致一些麻烦（甚至严重）的问题。

如果你的Linux机器已经用了好一段时间，你会发现你的home目录积累了大量的启动文件。他们通常会被称为dot文件，只因它们的文件名以点（dot）开头。它们很多是在某些程序初次启动时产生的，而一般人是不会去更改它们的。本章主要讲解shell启动文件。这些文件你很可能会修改甚至完全改写。现在来看看你需要为此花多少心思。

13.1　创建启动文件的规则

设计启动文件时，要考虑到用户。如果该机器只有你在用，那就不用太担心，因为有问题也只会影响到你，而且容易修复。但如果你创建的启动文件是该机器或该网络所有新用户的默认配置，或者说它会被复制到其他机器上使用，那么这设计任务就变得艰巨了。如果你的启动文件在10个用户那里出错，那你就要为此进行10次修复工作。

为别的用户创建启动文件时，必须要记住以下两点。

- 简单：启动文件的数量和内容都要尽可能地少，以使它们易守难攻。因为每多一个项目，就意味着多一个突破口。
- 可读：添加注释，方便用户了解文件的每一部分有什么用。

13.2　何时需要修改启动文件

在修改一个启动文件之前,先反省一下,真的非修改不可?以下才是修改启动文件的充分理由。

- 你想改变默认提示。
- 你需要提供一些关键的本地安装软件。(但还是建议你先考虑使用wrapper脚本。)
- 现存的启动文件有错漏。

如果本来你的Linux运作正常，那更要小心了。另外，有些默认的启动文件会与/etc里的其他文件交互。

虽然说了这么多，但如果你对修改默认配置没有兴趣，那么你是不需要阅读本章的，所以我们只挑重要的内容讲。

13.3　shell 启动文件的元素

shell启动文件里该有什么？明显地，需要有命令路径和提示符的配置。但命令路径和提示符具体应该是怎样的呢？还有，启动文件里放多少东西才算是过多呢？

接下来的几个小节将讨论shell启动文件的必要元素——路径、提示符、别名以及权限掩码。

13.3.1　命令路径

shell启动文件中最重要的就是命令路径。该路径应包含用户要用到的所有应用程序所在的目录。至少，按照顺序应包含这些：

```
/usr/local/bin
/usr/bin
/bin
```

这个顺序可确保你能使用/usr/local中的特定变体来覆盖默认程序。

大多数Linux发行版中，几乎所有软件包都会被安装到/usr/bin中。但还要留意一些少数情况，例如游戏会放在/usr/games中，而图形应用又放在另外的地方。所以先检查下你系统的默认设置，然后确保所有常用的程序都能在以上目录找到。如果找不到，那么你的系统可能就没法用了。不要为了迁就新装软件的目录而改变默认的命令路径。为了适应分散的软件目录，你可以用一个简单的方法，就是将那些软件的符号链接放到/usr/local/bin中。

很多人喜欢在自有的bin目录中放置shell脚本和程序,所以你可以将以下这行放在命令路径的开头：

```
$HOME/bin
```

注解　有种新的惯用方法是将二进制文件放在$HOME/.local/bin中。

如果你对系统工具感兴趣（例如traceroute、ping和lsmod），可以为路径加上sbin目录：

```
/usr/local/sbin
/usr/sbin
/sbin
```

在命令路径里加点（.）

命令路径中还有个颇有争议性的小元素：点（.）。命令路径中有了点的话，你就不用在程序名前加./即能直接运行当前目录中的程序。这看上去很便于写脚本和编译程序，但其实是有问题的，主要有以下两个原因。

- 可能会导致安全问题。永远不要将点放在命令路径的前端。考虑一下可能会发生的后果：黑客将木马打包成一个叫ls的文件，散布于网上。就算你将点放在命令路径的末端，你也可能因为打错sl或ks而触发意外。
- 可能产生不一样的结果，令人混淆。当前目录的切换可能会导致命令相同但效果不同。

13.3.2　帮助手册的路径

传统的帮助手册路径来自于环境变量MANPATH，但你不应该设置它，因为这样会重写了/etc/manpath.config的默认设置。

13.3.3　提示符

有经验的用户都不想看到冗长、复杂、无用的提示符。相比之下，很多管理员和发行版却喜欢将所有东西都拖拽到默认的提示符中。你做出的选择应该迎合用户的需要。如果他们真的需要当前目录、主机名、用户名，则可以将它们加入到提示符中。

最重要的是，不要用到shell的关键字，例如：

```
{ } = & < >
```

注解　要特别小心“>”字符。如果你拿shell窗口的内容来复制粘贴，可能会导致你当前目录中出现空文件（回忆一下“>”的重定向输出功能）。

其实默认的shell提示符也不是很理想的。例如，默认的bash提示符包含了shell名称和版本号。以下这个简单的bash提示符设置以常见的$结尾（传统的csh提示符则以%结尾）：

```
PS1='\u\$ '
```

\u用于代替当前用户（详见bash(1)帮助手册的PROMPTING节）。

其他常用的替代选项如下所示。

- \h：主机名（不含域名的短形式）。
- \!：历史号。
- \w：当前目录。因为这可能很长，你可以用\W来只显示当前路径末端的目录。
- \$：普通用户显示为$，root用户显示为#。

13.3.4　别名

别名是用户环境的难点之一。它是shell的一种特性，能在执行命令之前替换其中的字符串，可作为一种减少打字量的途径。但其实它也是有缺点的，如下所列。

- 它可以连参数也替换掉。
- 它容易让人困惑。shell的内置命令which可以告诉你一个东西它是不是别名，但不能告诉你哪里定义了这个别名。
- 它在子shell和非交互式shell中无法使用，只能在当前shell中使用。

鉴于以上缺点，你应该尽可能避免使用别名，因为编写shell函数或全新的shell脚本会更好控制。现代计算机都能快速地启动和运行shell，所以别名与全新的命令用起来几乎没有差异。

然而，当你仅想在某部分shell环境中改变命令意义时，可使用别名（或函数）。因为shell脚本是无法改变环境变量的，它运行于子shell中。

13.3.5　权限掩码

如第2章所述，你可以用shell的内置命令umask（权限掩码）来设置默认权限。你应该在某个启动文件中使用umask，以确保无论你用什么程序来创建文件，都能给该文件赋予你需要的权限。以下是两个值得参考的掩码选择。

- 077：此掩码最严格，它使得新建的文件与目录只能由创建者访问。在多用户的系统中，如果你不想其他人查看你的文件，那么是可以用这个的。然而，如果默认设成这样，而你的用户又不懂怎样自行修改权限的话，那么他们在分享文件的时候就会出现问题。（经验不足的用户可能会将文件修改成所有人可写。）
- 022：此掩码能让其他用户查看新建的文件和目录的内容。这在单用户的环境中也是有意义的，因为很多守护进程是以伪用户的形式运行的，它们不能访问以077掩码创建的文件和目录。

注解　某些应用（特别是邮件应用）会将掩码重写成077，因为这些应用认为由它们创建的文件只应由创建者查看。

13.4　启动文件的顺序及例子

在了解了启动文件该有的内容后，现在来看下具体的例子。令人意外的是，创建启动文件的

难点之一，竟是选择使用哪些启动文件。下一节我们会介绍Unix shell中两个最流行的做法：bash和tcsh。

13.4.1 bash shell

在bash里，有.bash_profile、.profile、.bash_login和.bashrc等启动文件名可供选择。那么到底哪个适合设置命令路径、手册路径、提示符、别名和权限掩码呢？答案就是：你应该使用.bashrc，并建立名为.bash_profile的符号链接指向它，因为bashshell的实例是有不同类型的。

shell实例的类型主要有两种：交互式和非交互式。但其中我们只关心交互式的，因为非交互式的shell（例如运行shell脚本的那些）通常不读取启动文件。而交互式shell则是从终端读取并运行命令的环境，本书中曾提到过。它分为登录shell和非登录shell两种。

登录shell

按照传统说法，登录shell就是你通过/bin/login之类的程序登录系统时所获得的shell。用SSH来远程登录，获取的也是登录 shell。登录shell的基本概念就是，它是一个初始shell。可以用echo $0来验证一个shell是不是登录shell，如果它打印出的第一个字符是单杠（-），那当前shell就是登录shell。

当bash以登录shell的形式运行时，它会运行/etc/profile。然后就会寻找用户的.bash_profile、.bash_login和.profile文件，并运行它所找到的第一个文件。

让一个非交互式shell像登录shell一样运行启动文件，也是可以的，虽然听起来有点怪。要想这样做，只要在启动该shell时带上-l或--login选项即可。

非登录shell

非登录shell就是登录以后另外再运行的shell。即交互式shell中，除登录shell之外的shell。窗口化的系统终端程序（如xterm、GNOME终端等）都会启动非登录shell，除非你指定它提供登录shell。

当bash以非登录shell的形式启动时，它会运行/etc/bash.bashrc以及用户的.bashrc。

两种shell所带来的影响

之所以有两种shell，是因为以前人们通过终端登录时使用的是登录shell，而登录后运行窗口化程序或screen程序的子shell则是使用非登录shell。如果在所有非登录的子shell中都运行用户环境的设定程序，那是很浪费资源的。所以，你可以在登录shell中运行包含各种各样启动命令的.bash_profile，而让非登录的.bashrc文件只包含别名设定和其他“轻量”的东西。

今时今日，大多数的桌面用户都是通过图形界面管理器（下一章你就会学到）登录的。它们大多数是以一个非交互式的登录shell启动，以适配上述的登录与非登录模型。如果不这么做的话，那你就要在.bashrc中做好所有的环境设定（命令路径、手册路径等等），不然你的shell终端窗口将没有你的个人设定。如果你想通过控制台或远程登录来登录系统，那么你就还需要有.bash_profile，因为登录shell是不会读取.bashrc的。

.bashrc例子

为了同时满足非登录和登录shell，你会如何创建一个能被当作.bash_profile使用的.bashrc呢？以下是一个初级的例子（其实也足够完美了）。

13

```
# Command path.
PATH=/usr/local/bin:/usr/bin:/bin:/usr/games
PATH=$HOME/bin:$PATH

# PS1 is the regular prompt.
# Substitutions include:
# \u username \h hostname \w current directory
# \! history number \s shell name \$ $ if regular user
PS1='\u\$ '

# EDITOR and VISUAL determine the editor that programs such as less
# and mail clients invoke when asked to edit a file.
EDITOR=vi
VISUAL=vi

# PAGER is the default text file viewer for programs such as man.
PAGER=less

# These are some handy options for less.
# A different style is LESS=FRX
# (F=quit at end, R=show raw characters, X=don't use alt screen)
LESS=meiX

# You must export environment variables.
export PATH EDITOR VISUAL PAGER LESS

# By default, give other users read-only access to most new files.
umask 022
```

如早先提到的，你可以建立名为.bash_profile、指向.bashrc的符号链接，以共用.bashrc的内容，或者直接在.bash_profile中只写这么一行：

```
. $HOME/.bashrc
```

检查是否是登录和交互式shell

当.bashrc与.bash_profile一致时，一般不用再让登录shell执行什么额外的命令了。然而，如果你还想为登录和非登录的shell定义不同的动作，那么可以在.bashrc中加入以下测试，它会检查$-变量是否包含i字符：

```
case $- in
 *i*) # interactive commands go here
    command
    --snip--
    ;;
 *)   # non-interactive commands go here
    command
    --snip-
    ;;
esac
```

13.4.2 tcsh shell

实际上，所有Linux系统标配的csh都是tcsh。它是C shell的加强版，突出了命令行编辑、多模式文件名、命令补全等特性。即使你没将tcsh作为新用户的默认shell，你也应该提供tcsh的启动文件，以防你遇到想用tcsh的人。

在tcsh中，你不用担心登录和非登录的区别。它启动的时候，会先查找.tcshrc文件，如果没找到，就会去找csh的启动文件.cshrc。之所以按照这个顺序，是因为.tcshrc可以放置csh不支持的命令。尽管没什么人会用csh，但你还是应该坚持使用.cshrc，而非.tcshrc。如果碰巧遇到有使用csh的，那么幸好.cshrc还能运作。

.cshrc例子

以下是.cshrc的一个例子。

```
# Command path.
setenv PATH /usr/local/bin:/usr/bin:/bin:$HOME/bin

# EDITOR and VISUAL determine the editor that programs such as less
# and mail clients invoke when asked to edit a file.
setenv EDITOR vi
setenv VISUAL vi

# PAGER is the default text file viewer for programs such as man.
setenv PAGER less

# These are some handy options for less.
setenv LESS meiX

# By default, give other users read-only access to most new files.
Umask 022

# Customize the prompt.
# Substitutions include:
# %n username %m hostname %/ current directory
# %h history number %l current terminal %% %
set prompt="%m%% "
```

13

13.5 用户默认设置

为新用户编写启动文件和选择默认设置的最佳方法，就是以新用户的身份来试用一个系统。即新建测试账号，并赋予其空的home目录，从头开始写启动文件，而非复制你home目录的启动文件。

当你认为你写好的时候，就用测试账号以各种方式（控制台、远程等）登录。确保自己测试到尽可能多的内容，包括窗口化系统操作和帮助手册查阅。当你觉得操作顺畅时，就创建另一个测试账号，并从之前的测试账号那里复制启动文件。如果依然可行，那么你这套新的启动文件就可以发布使用了。

接下来的部分概述了一些可作参考的新用户默认设置。

13.5.1　shell默认设置

Linux上的默认shell应该是bash，因为：

- 人们惯用bash环境来编写脚本（csh不适合写脚本），所以交互式环境也该用bash；
- Linux标配bash；
- bash用到了GNU readline，因此它的接口与其他工具是相同的；
- bash的I/O重定向和文件操作很易上手。

然而，有很多Unix老手是只用csh和tcsh的，因为他们不想改变习惯。当然，你也可以按自己喜好来选择用哪种shell。如果没有特殊喜好，那就用bash，并把新用户的默认shell设置为bash。（用户可以用chsh命令来切换到自己喜欢的shell。）

注解　shell的种类还有很多（rc、ksh、zsh、es等）。有些是不适合新手的，但zsh和fish是比较受新用户欢迎的两个替代选择。

13.5.2　编辑器

按照传统，默认编辑器是vi或emacs。Unix系统中一般都有它们的存在，且这么常见就意味着它们是没什么问题的。不过很多Linux发行版则会将nano设定为默认编辑器，因为它对于新手更易用。

但在启动文件中，应该避免大量的编辑器设置。在.exrc中写上set showmatch是没有坏处的，但最好别写上showmode、自动缩进或右边距之类能明显改变编辑器行为或外观的设置。

13.5.3　翻页器

PAGER环境变量设为less是不会有错的。

13.6　启动文件的一些陷阱

在shell启动文件中，务必注意以下事项：

- 不要在启动文件中放置任何图形命令；
- 不要在启动文件中设置环境变量DISPLAY；
- 不要在启动文件中设置终端类型；
- 不要在启动文件中吝啬于注释；
- 不要在启动文件中打印东西到标准输出；
- 不要在启动文件中设置LD_LIBRARY_PATH（详见15.1.4节）。

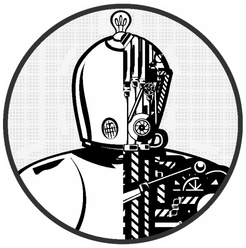
13.7　前瞻

因为本书只讲解Linux底层，所以不会涉及窗口环境的启动文件。那当然是一个很大的话题，因为供你登录用的显示管理器也有自己的一套启动文件，诸如.xsession、.xinitrc，以及无尽的GNOME和KDE相关的项目组合。

选择窗口启动文件看起来令人眼花缭乱，而且没有通用标准。下一章就会讲到各种可能性。当你决定好系统的用途时，你可能会大动那些图形相关的环境设定。这无所谓，但千万别把改动套到新用户身上。“保持简单”这一宗旨不仅适用于shell启动文件，也适用于GUI启动文件。事实上，你甚至可能完全不需要修改GUI启动文件。

第 14 章

Linux桌面概览



本章会简单介绍一些Linux桌面系统中的常见组件。Linux的各种应用之中，桌面应该算是最丰富多彩的一种，因为它选择众多，而大多数发行版也一般都会提供桌面系统供用户使用。

跟Linux的其他部分（如存储和网络）不同，桌面结构没有太多层级。桌面的每个组件都只对应特定的任务，只在必要时才与其他组件沟通。有些组件会共享通用的库（特别是图形工具库），你可以把它看作抽象的层次，也就是它所能到达的深度。

大致上，本章宏观地讨论各个桌面组件，其中两个会讲得比较细：X Window系统（它是大多数桌面的核心设施），以及D-Bus（一种应用于系统各部分的进程间通信服务）。动手实践也只限于一些诊断工具，虽然它们不常用到（因为大部分GUI是不需要用shell命令来交互的），但会帮助你了解系统的底层机制，或许还能在探索的路上为你提供一些乐趣。我们还会简单讲讲打印。

14.1　桌面组件

Linux桌面配置有很大的灵活性。用户体验（桌面的外观及给人的感觉）基本来自各种应用或应用中的组件。如果你不喜欢某个应用，你可以找个替代品。如果找不到，你还可以自己写一个。Linux开发者往往有不同喜好，这使得开发出来的产品有很多花样。

为了能协同工作，各种应用就需要有一些共性，而在几乎所有的Linux桌面组件中，这种共性就是X服务器（即X Window系统服务器）。你可以把它想象为桌面的“内核”，管理着窗口功能和显示配置，并处理来自键盘和鼠标等设备的输入。它是难以取代的（详见14.4节）。

X服务器只是一个服务器，它无法决定桌面的表现形式。相反，用户界面是由X客户端处理的。基本的X客户端应用如终端窗口和网页浏览器，会连接到X服务器，并请求它绘制窗口。这样X服务器就会算出窗口的位置并在该位置提供图形。X服务器也会在适当的时候将用户输入反馈给客户端。

14.1.1 窗口管理器

X客户端不一定是窗口化的应用，它可以是其他客户端的服务供应者，或者提供接口功能。而*窗口管理器*或许是最重要的服务于客户端的应用，因为它负责窗口的位置安排，以及提供一些交互式装饰（例如用于窗口移动、缩放的标题栏）。这些都是用户体验的核心。

窗口管理器的实现有很多种。像Mutter/GNOME Shell和Compiz之类基本上是独立的窗口管理器，而Xfce之类则是内置于整个环境中的。大部分窗口管理器的目标都是方便用户使用。有一些是为了实现特别的视觉效果，或者只为提供极简的界面。窗口管理器一般是不会有标准的，因为用户的品味和需求多种多样，而且经常变化，所以也经常诞生新的窗口管理器。

14.1.2 工具包

桌面应用都有一些相似的元素，例如按钮、菜单，这些都叫*部件*。为了加快应用开发的速度，以及提供一个符合人们习惯的界面，程序员们都会使用*图形工具包*来制作这些元素。在Windows或Mac OS X这类操作系统中，供应商会提供一套通用的工具包，大多数程序员都会使用到。而Linux最常用的则是GTK+工具包。除此之外，Qt框架等其他工具也不少见。

工具包一般会包含共享库和支持文件，如图像和主题信息。

14.1.3 桌面环境

尽管工具包能提供统一的外观，但有些桌面的细节还是需要不同应用进行某种程度的协作才能表现的。比如说，你可能想让某个应用与另一个应用共享数据，或者刷新桌面上共用的通知栏。若要满足这一需要，可以将工具包和其他库绑在一起，放到一个叫作*桌面环境*的大包中。GNOME、KDE、Unity和Xfce都是常见的Linux桌面环境。

工具包在多数桌面环境中都处在核心位置，但要创建一个统一的桌面，环境还必须具备各种支持文件，例如图标和配置所构成的主题。所有这些都要被描述设计协议（如应用菜单和标题如何呈现、应用对某些系统事件要做什么反应等）的文档捆绑在一起。

14.1.4 应用

桌面的顶端就是各种应用了，诸如网页浏览器、终端窗口等。原始如xclock程序，复杂如Chrome浏览器和LibreOffice套件，都是X应用。一般情况下它们是独立工作的，但其实它们也会使用进程间通信来响应与它们有关的事件。例如，一些应用会对以下情况做出反应：挂载了新的存储设备、收到新邮件或即时信息。这些通信一般发生在D-Bus上，我们会在14.5节讲到。

14

14.2 近观 X Window 系统

X Window系统（http://www.x.org/）历来就很庞大，它基础的发行版包括了X服务器、客户端支持库和各种客户端。因为GNOME和KDE等桌面环境的出现，X发行版的角色也一直在变换。

现在它的关注点主要在核心服务器（即管理渲染和输入设备的部分）和简化的客户端库。

X服务器的运行不难识别。它就叫X。看看进程列表，你通常会发现它以这些选项运行着：

```
/usr/bin/X :0 -auth /var/run/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
```

这里的:0被称为*显示接口*，是一个标识，代表着一个或多个用键盘和（或）鼠标访问的显示器。通常，显示接口只与单个显示器对应，但你也可以将多个显示器放到一个显示接口上。使用X会话时，环境变量DISPLAY包含了显示接口的标识。

注解 显示接口可以细分为多个屏幕，例如:0.0和:0.1，但这种做法越来越少见了，因为X扩展（如RandR）能将多个显示器合并成一个大的虚拟屏幕。

Linux的X服务器是运行在虚拟终端的。上例的vt7参数表明了它运行在/dev/tty7之上。（一般服务器会优先选择第一个可用的虚拟终端。）你可以在不同的虚拟终端上同时运行不同的X服务器，这样每个服务器都会需要一个各自独立的显示接口标识。你可以用CTRL-ALT-FN键或chvt命令在不同服务器之间切换。

14.2.1 显示管理器

一般你是不会从命令行启动X服务器的，因为这么做不会启动任何客户端来连接到这个服务器，结果只会得到一个空白的屏幕。相反，通常的做法是用*显示管理器*来启动X服务器，它会在屏幕上放置一个登录框。当你登录之后，显示管理器就会启动一系列的客户端诸如窗口管理器和文件管理器，以便你使用机器。

显示管理器有很多种，例如gdm（用于GNOME）和kdm（用于KDE）。上例中的lightdm是一个跨平台的显示管理器，它能打开GNOME和KDE会话。

若想从虚拟控制台而非显示管理器开启X会话，你可以运行startx或xinit命令。然而，这样获取的会话相当简单，与显示管理器的会话完全不同，因为它们的机制以及启动文件都不同。

14.2.2 网络透明性

X还有一个特性就是其网络透明性。因为客户端跟服务器的沟通是遵循协议的，所以我们可以让服务器监听6000端口的TCP连接。也就是说，可以通过网络来连接不同机器上的客户端和服务器。客户端只需通过验证，就可将窗口信息发送给服务器。

不幸的是，这个方法是没有加密的，所以并不安全。为了填补这个漏洞，大多数发行版都会关闭X服务器的网络监听器（如上例的--nolisten tcp选项）。不过，如第10章提到的，你还是可以通过SSH管道，使用Unix域套接字让X客户端远程连接X服务器。

14.3　探索 X 客户端

虽然一般人不会从命令行的角度来思考GUI的运作，但还是有一些工具这么做。借助它们，你可以监控客户端的运行。

xwininfo是最简单的工具之一。不带参数运行它的话，它会要你点选一个窗口：

```
$ xwininfo
xwininfo: Please select the window about which you
          would like information by clicking the
          mouse in that window.
```

选完之后，它就会打印出该窗口的信息列表，如位置和尺寸：

```
xwininfo: Window id: 0x5400024 "xterm"

  Absolute upper-left X: 1075
  Absolute upper-left Y: 594
--snip--
```

注意这里的窗口ID，它是X服务器和窗口管理器用来识别窗口的标识。xlsclients -l命令可打印出所有窗口ID的客户端。

注解　有一种特别的窗口，叫root窗口，是画面的背景。然而，你一般是不会看到它的（详见14.3.2节）。

14.3.1　X 事件

X客户端通过事件系统获取输入和服务器状态等信息。X事件的工作方式类似于其他异步进程间通信（如udev事件和D-Bus事件）：X服务器从输入设备等源头获取信息，再将它们当作事件重新分发给感兴趣的X客户端。

你可以通过运行xev命令来体验什么叫作事件。运行xev，就会打开一个可以打字、点击和使用鼠标的窗口。根据你的操作，xev就会输出X服务器接收到的事件的描述。以下是一个鼠标事件的输出例子：

```
$ xev
--snip--
MotionNotify event, serial 36, synthetic NO, window 0x6800001,
    root 0xbb, subw 0x0, time 43937883, (47,174), root:(1692,486),
    state 0x0, is_hint 0, same_screen YES

MotionNotify event, serial 36, synthetic NO, window 0x6800001,
    root 0xbb, subw 0x0, time 43937891, (43,177), root:(1688,489),
    state 0x0, is_hint 0, same_screen YES
```

注意括号中的坐标。第一个坐标代表了窗口中鼠标的位置，而第二个（root:）则是鼠标在整个屏幕中的位置。

其他低层次的事件还包括键盘敲击和按钮点击，而鼠标进入或离开窗口、窗口获得或失去窗口管理器的关注则属于高级事件。以下就是对应的离开事件和失去关注事件：

```
LeaveNotify event, serial 36, synthetic NO, window 0x6800001,
    root 0xbb, subw 0x0, time 44348653, (55,185), root:(1679,420),
    mode NotifyNormal, detail NotifyNonlinear, same_screen YES,
    focus YES, state 0

FocusOut event, serial 36, synthetic NO, window 0x6800001,
    mode NotifyNormal, detail NotifyNonlinear
```

xev的一个常见用法是从不同的键盘抽取键码和键标志，并对它们重新映射。以下是敲击L键的输出情况，其键码是46：

```
KeyPress event, serial 32, synthetic NO, window 0x4c00001,
    root 0xbb, subw 0x0, time 2084270084, (131,120), root:(197,172),
    state 0x0, keycode 46 (keysym 0x6c, l), same_screen YES,
    XLookupString gives 1 bytes: (6c) "l"
    XmbLookupString gives 1 bytes: (6c) "l"
    XFilterEvent returns: False
```

你还可以加上-id id（其中id是xwininfo输出的ID）选项，让xev与某个现有的窗口ID关联，或者用-root选项监视root窗口。

14.3.2　理解 X 输入以及偏好设定

X最让人困惑的地方或许是它提供多种途径设定偏好，但不是所有的都可行。例如，有种常见的键盘偏好是将Caps Lock键映射到Control键。它的实现方法有很多种，可用古老的xmodmap命令进行微小的调整，也可用setxkbmap工具进行完全的重映射。但你怎么知道该用哪个呢？你需要知道系统的哪一块负责这个问题，但这有些困难。不过至少要记住，桌面环境是可以有自己的设定以及重写的。

总结起来，底层设施有以下几点是需要关注的。

输入设备（通用）

X服务器使用X输入扩展来管理各种不同设备的输入。基本的输入设备有两种——键盘和指针（鼠标）——其实你可以想用多少设备就用多少。为了做到同时使用多个同种设备，X输入扩展会创建一个“虚拟核心”设备，用于将输入汇集到X服务器。这个虚拟核心设备就被称为主机（master），各种接入到机器的物理设备就被称为从机（slave）。

查看机器上的设备配置，可用xinput --list命令：

```
$ xinput --list
  Virtual core pointer                    id=2    [master pointer  (3)]
      Virtual core XTEST pointer          id=4    [slave  pointer  (2)]
      Logitech Unifying Device            id=8    [slave  pointer  (2)]
  Virtual core keyboard                   id=3    [master keyboard (2)]
      Virtual core XTEST keyboard         id=5    [slave  keyboard (3)]
      Power Button                        id=6    [slave  keyboard (3)]
      Power Button                        id=7    [slave  keyboard (3)]
      Cypress USB Keyboard                id=9    [slave  keyboard (3)]
```

每个设备都有一个ID，可用于xinput和其他命令。上例中2和3这两个ID属于核心设备，而8和9则是真实设备。注意，电源键也是X输入设备。

大多数X客户端只会监听核心设备的输入，而对其他具体设备发起的事件却漠不关心。事实上，大多数客户端完全不知道X输入扩展的存在。不过，客户端其实可以通过输入扩展来识别出某个具体设备。

每个设备都有自己的属性。使用xinput时带上设备的号码，就可查看其属性，如下所示。

```
$ xinput --list-props 8
Device 'Logitech Unifying Device. Wireless PID:4026':
        Device Enabled (126): 1
        Coordinate Transformation Matrix (128): 1.000000, 0.000000,0.000000,
0.000000, 1.000000, 0.000000, 0.000000, 0.000000, 1.000000
        Device Accel Profile (256):     0
        Device Accel Constant Deceleration (257):       1.000000
        Device Accel Adaptive Deceleration (258):       1.000000
        Device Accel Velocity Scaling (259):    10.000000
--snip--
```

如你所见，它能打印出一些有趣的属性，这些属性都能使用--set-prop选项来更改（详见xinput(1)帮助手册）。

鼠标

你可以用xinput命令修改设备相关的属性，其中最有用的就是鼠标（指针）。很多设置都可以通过直接修改属性来做到，但其实还有更简单的方法，就是使用xinput的--set-ptr-feedback和--set-button-map选项。例如你有一个三键的鼠标dev，你想改变其按钮的操作顺序（这比较适合惯用左手的用户），那可以试试以下命令：

```
$ xinput --set-button-map dev 3 2 1
```


键盘

键盘布局的多样化使得我们无法将所有布局都整合到X中。所以它的协议中内置了键盘映射的功能，让你可以通过xmodmap命令来定义布局。但为了更方便操控，大部分现代系统其实也都是用XKB（X键盘扩展）的。

XKB很复杂，所以很多人为了顺手还是使用xmodmap。XKB的基本思想是让你定义一个键盘

映射，然后用xkbcomp命令将它编译，最后用setxkbmap命令将它载入到X服务器中并激活使用。该系统有以下两个特别有趣的特性。

- 你可以只定义一部分，以补充现存的映射。这种做法有助于将Caps Lock键变成Control键之类的任务，很多桌面环境中的图形化的键盘偏好工具都是这么做的。
- 你可以为各个接入的键盘定义各自的映射。

桌面背景

X有个旧的命令xsetroot，它可以让你设置root窗口的背景色和其他属性，但在大多数机器上它并不可行，因为root窗口是不可见的。相反，大部分桌面环境却会将一个大窗口放置在其他所有窗口的背后，以在其中实现"动态壁纸"或桌面文件浏览的功能。通过命令行更换背景的方法有很多（例如某些GNOME的gsettings命令），但真做起来是很费时间的。

xset

最古老的偏好设定命令或许是xset。它已经很少用了，但xset q命令确实能快速显示出一些功能的状态。也许其中最有用的选项是关于屏幕保护和显示器电源信号管理（Display Power Management Signaling，以下简称DPMS）的设定。

14.4　X 的未来

在以上小节的阅读过程中，也许你会觉得X是个很老的系统，它被改来改去以适应新的需求。其实它并不是很古老。X系统首创于20世纪80年代，尽管在这些年里的变化非常大（它原本的设计就很注重灵活性），但它未来不会有太大变化了。

X的发展有个很明显的迹象，即它的服务器支持了极多的库，其中很多是为了向下兼容。但更明显的是，用服务器管理客户端、客户端的窗口，并作为它们与内存的中介，这种做法对性能有很大影响。更高效的做法是用一种轻量的组合窗口管理器来管理窗口，并只对显存做少量控制，且允许应用自行在显存中渲染窗口的内容。

Wayland是基于这种做法的新标准，它已经开始引人注目。它最突出的部分是客户端与组合窗口管理器的沟通协议。另外，它还定义了输入设备管理和X兼容系统。Wayland协议同样支持网络透明性。现在Linux桌面组件如GNOME和KDE都支持Wayland。

但X的替代者不止Wayland一个，还有Mir，不过它的实现跟Wayland有点不同。某种程度上来说，肯定会有至少一种系统成为主流，它可能在Wayland和Mir之中，也可能不在。

这些新产品之所以出名，是因为它们不只用于Linux桌面。X Window系统的低效使其不适合平板电脑和智能手机，于是制造商用其他实现来驱动Linux嵌入式显示器。只不过，标准化的直接渲染更具成本效益。

14.5　D-Bus

D-Bus（即桌面总线）是Linux桌面系统的最重要的产物之一，它是一个消息传递系统。D-Bus之所以重要，是因为它作为一种进程间通信的机制，使得各种桌面应用能够相互沟通。同时，大

多数的Linux系统都是用它来把系统事件（例如插入USB设备）通知给进程的。

D-Bus本身包含有支持任何两个进程相互沟通的库，其中定义了规范进程间通信的协议。该库其实只是一种进程间通信方式，就像Unix域套接字。它的重点是一个叫dbus-daemon的"中央槽"。需要对某些事件做出反应的进程，可以到dbus-deamon上注册，然后就能收到想要的事件通知了。当然，进程也可以创建事件。例如，udisks-daemon进程从ubus监听硬盘事件，并发送到dbus-deamon，而dbus-deamon会把这些事件再转发给那些对硬盘事件感兴趣的应用。

14.5.1 系统和会话实例

D-Bus在Linux中正变得越来越重要，而且它的用途不只在桌面。systemd和Upstart也使用它来通信。然而，在核心系统中加入对桌面工具的依赖，这有违Linux的设计宗旨。

为了解决这个问题，我们将dbus-daemon实例（进程）分为两种。一种叫系统实例，它在开机时由init启动，并带上`--system`选项。这种实例通常作为D-Bus用户来运行，它的配置文件是/etc/dbus-1/system.conf（一般你不应该修改这个文件）。进程可以通过/var/run/dbus/system_bus_socket的Unix域套接字连接到该实例。

另一种叫会话实例。与系统实例不同的是，会话实例只在你打开桌面会话时才会运行。你运行的桌面应用会连接这种实例。

14.5.2 监视 D-Bus 消息

查看系统实例与会话实例区别的最佳方法之一是监视总线上的消息。试试使用`dbus-monitor`的`system`模式：

```
$ dbus-monitor --system
signal sender=org.freedesktop.DBus -> dest=:1.952 serial=2 path=/org/
freedesktop/DBus; interface=org.freedesktop.DBus; member=NameAcquired
   string ":1.952"
```

以上启动信息说明了该监视器已连接，并获得了一个名字。这样运行的话，能看见的信息并不多，因为系统实例一般不太繁忙。想多看点信息的话，插入一个USB存储设备试试。

相比之下，会话实例就比较忙了。假设你登录了一个桌面会话，然后运行以下命令：

```
$ dbus-monitor --session
```


接下来在不同窗口上点击鼠标。如果你桌面用到了D-Bus，你会看到有大量的信息输出，显示了那些被激活的窗口。

14.6 打印

在Linux上打印文档是一个多步的过程，其步骤如下所示。

(1) 打印程序通常会先将文档转成PostScript格式。不过也可以不这么做。

(2) 程序将文档发给打印服务器。

(3) 打印服务器收到文档后，将其放到打印队列中。

(4) 当轮到该文件时，打印服务器会将其发送到打印过滤器。

(5) 如果发现该文档不是PostScript格式，打印过滤器可以对其进行转换。

(6) 如果目标打印机不能识别PostScript，打印机驱动会将该文档转换成打印机能识别的格式。

(7) 打印机驱动可在文档上加一些额外的指令，例如纸匣和复件数。

(8) 最后打印服务器将文档发给打印机。

这里面最让人困扰的，就是要在PostScript上绕来绕去。其实，PostScript是一种编程语言，所以如果你用它来打印文件，那么你实际上就是将一段程序发给了打印机。PostScript是类Unix系统中的打印标准，就像.tar是打包标准一样。（现在有些应用用到的PDF格式，也是能转成PostScript的。）

下文中会更详细地讲解打印格式，现在先看看队列系统。

14.6.1　CUPS

CUPS（http://www.cups.org/）是Linux和Mac OS X的标准打印系统。它的服务器守护进程是cupsd，你可以用`lpr`命令作为客户端来发送文件给这个守护进程。

CUPS有个突出的功能是实现了互联网打印协议（Internet Print Protocol，以下简称IPP），使得它允许客户端与服务器端通过TCP端口631进行类HTTP的事务处理。事实上，如果你系统上运行着CUPS，你就可以连接http://localhost:631/去看看你的打印配置和打印任务。大多数的网络打印机和打印服务器都支持IPP，就连Windows也是。IPP简化了建立远程打印机的任务。

通过网页界面来管理这个系统可能不太可靠，因为它的默认设置不太安全。作为替代方案，发行版中通常自带了图形界面工具以便你添加和更改打印机。这些工具会修改配置文件，它们一般位于/ets/cups中。因为配置文件可能比较复杂，所以最好还是用工具来处理。在出现问题的时候，用图形工具来新建打印机也是不错的做法。

14.6.2　格式转换与打印过滤器

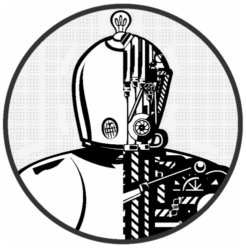
很多打印机，包括几乎所有低端型号的，都无法识别PostScript或PDF。为使Linux支持这些打印机，我们必须将文档转换成它们能识别的格式。CUPS把文档送给RIP（即光栅图像处理器）以生成位图。而RIP几乎总是使用Ghostscript（gs）程序来实现这个过程。但是，要让生成的位图能适应打印机的格式，还是有点麻烦的。所以，CUPS使用的打印机驱动会参考特定打印机的PostScript打印机定义（PostScript Printer Definition，以下简称PPD）文件，以解决分辨率和纸张大小之类的问题。

14.7　其他有关桌面的话题

Linux桌面系统的趣味性之一就在于它有很多项目供你选择。想了解不同的桌面项目，可参考http://www.freedesktop.org/提供的邮件列表和项目链接。你会发现桌面项目还有Ayatana、Unity、Mir等等。Linux桌面还有一大产品，那就是Chromium OS开源项目及其对应的Chromebook上的Google Chrome OS。本章谈到的很多桌面技术都在其中使用到了，只不过它是以Chromium/Chrome网页浏览器为核心的。Chrome OS也抛弃了很多传统桌面的东西。

第 15 章

开发工具



Linux和Unix在程序员中很受欢迎，这不只是因为它们工具和版本繁多，而且文档详尽、相当透明。不是只有程序员才能使用Linux的开发工具，但如果你使用的是Linux系统，那么了解它的工具还是有好处的。与其他操作系统不同，Unix的管理是很依赖于这些工具的。至少，你应该说得出它们的名字并知道怎样运行它们。

本章篇幅不长，但包含了大量的信息。不过你不必通晓全部，可以简单浏览一下，以后再回过头来看。其中共享库那一部分内容或许是最重要的。但要知道共享库的来历，你需要先知道如何创建程序。

15.1　C 编译器

了解如何运行C编译器能让你看透Linux上的程序的本质。大多数的Linux工具，以及很多Linux应用软件，都是用C或C++写成的。本章主要以C作为例子，但你把这些概念搬到C++上是没问题的。

C程序遵照传统的开发流程：写代码、编译代码、运行代码。也就是说，想让写好的C代码能运行起来，你必须先将其编译成计算机能理解的、低层次的二进制形式。你可拿C来跟后面讲到的一些脚本语言作对比，它们不需要编译。

注解　大多数发行版都默认不含有编译C的工具，因为它们比较占空间。如果你发现没有这些工具，对于Debian/Ubuntu，你可以安装build-essential包，而对于Fedora/CentOS，则可以用yum groupinstall。如果不行，就试着找一下“C compiler”。

虽然来自LLVM项目的新的C编译器clang越来越流行，但在大部分Unix系统上通行的C编译器还是GNU C编译器gcc。C源码的文件名以.c结尾。现在来看一个叫hello.c的独立的C源码文件，它来自Brian W. Kernighan和Dennis M. Ritchie的著作*The C Programming Language*, 2nd edition（Prentice Hall，1988）。

```
#include <stdio.h>

main() {
    printf("Hello, World.\n");
}
```

将以上代码置于一个叫hello.c的文件中，然后执行以下命令：

```
$ cc hello.c
```

这样会产生一个叫a.out的可执行文件，你可以像运行其他可执行文件一样运行它。不过，你若想给它另起一个名字（例如叫hello），可以带上-o选项：

```
$ cc -o hello hello.c
```

小程序一般这么做就可以了。可能你会想加入其他目录或者库（详见15.1.2节和15.1.3节），但在这之前，我们先看看一个稍微大点的程序。

15.1.1　多个源码文件

大部分C程序都写得很大，不宜放到一个单独的源码文件中。大文件不便于程序员管理，编译时也可能会出错。因此，开发者会将代码分块放在多个文件中。

我们不会将这些.c文件马上编译成可执行文件，而是使用编译器的-c选项来给每个文件生成对应的对象文件。假设你有main.c和aux.c两个文件，以下两条编译器命令会完成建立程序的大部分工作：

```
$ cc -c main.c
$ cc -c aux.c
```

以上两命令会从这两个文件编译出main.o和aux.o两个对象文件。

对象文件是一种二进制文件，除了个别地方，处理器差不多能解读它。首先，操作系统是不知道如何运行对象文件的；其次，你可能会需要将一些对象和系统库组合成一个完整的程序。

要从一个或多个对象文件建立一个功能完整的可执行程序，你需要用连接器，如Unix中的ld命令。但程序员是很少在命令行上用它的，因为C编译器包含了该步骤。所以，想从上面两个对象文件建立myprog的话，就用以下命令来连接它们：

```
$ cc -o myprog main.o aux.o
```

虽然这样可以手动编译多个源码文件，但正如上例所示，若文件太多的话，编译过程还是很难管理的。15.2节讲到的make系统是传统Unix上标准的编译管理器。要管理下两节提到的那些文件，这个系统尤为重要。

15.1.2　头（include）文件和目录

C的头文件是用于保存类型和函数声明的附加文件。例如，stdio.h就是一个头文件（见15.1节中的小程序）。

不幸的是，有很多编译问题是与头文件有关的。大多数情况下是因为编译器找不到头文件或库。有些情况下是因为程序员忘了包含必要的头文件，导致某些代码编译不通过。

修复include文件的问题

记住正确的include文件并不总是易事。有时相同名字的include文件放在了不同的目录中，令人难以分辨哪个才是需要的。当编译器找不到include文件时，就会出现类似这样的报错信息：

```
badinclude.c:1:22: fatal error: notfound.h: No such file or directory
```

该信息说明了badinclude.c文件需要参考的notfound.h头文件无法找到。这个错误源自badinclude.c的第一行：

```
#include <notfound.h>
```

Unix默认的include目录是/usr/include，编译器一般就看那里，除非你指定它看别的地方。（大多数包含头文件的路径都会带有include的名字。）

> **注解**　在第16章你会学到更多有关如何寻找丢失的include文件的内容。

假设notfound.h在/usr/junk/include中，你可以加上-I选项，让编译器看到这个目录：

```
$ cc -c -I /usr/junk/include badinclude.c
```

现在就不会因为头文件的引用出错而不能编译了。

另外，你还要注意include中双引号（""）与尖括号（<>）的区别：

```
#include "myheader.h"
```

双引号意味着头文件不在系统的include目录中，需要编译器从其他地方寻找。这通常表示它与源码文件处于同一目录中。如果你使用双引号时出现问题，可能是你要编译的程序并不完整。

什么是C预处理器

其实，并不是C编译器去寻找include文件，而是C预处理器（C Preprocessor，简称cpp）。它

是编译器在解析程序之前先在源码上运行的一个东西。预处理器会将源码重写成一种编译器能理解的形式，它能使源码更易读（并提供捷径）。

源码中的预处理器命令叫作指令（directive），它们以#开头，分为以下三种。

- include文件：#include指令会使预处理器将整个文件包含进来。例如在上节中提到的，是编译器的-I选项让预处理器在指定目录中搜索include文件。
- 宏定义：#define BLAH something这样的一行，会使预处理器将源码中所有的BLAH替换成something。一般我们约定宏的名字都是大写的，但有人会将宏的名字起得像函数名或变量名。（这一直都很让人头痛，更有许多程序员对滥用预处理器的现象嗤之以鼻。）
- 条件：你可以用#ifdef、#if和#endif来对代码进行分块。#ifdef MACRO指令用于检查宏MACRO是否已定义，而#if condition则检查condition是否非零。当预处理器发现if语句后的条件为false时，它就不会将#if和#endif之间的代码交给编译器。如果你想看懂C程序，最好先习惯这种指令。

注解　你也可以不在源码中定义宏，而用编译器的-D选项来实现：-DBLAH=something。它的效果就如上面的#define BLAH something。

下面有一个条件指令的例子。当预处理器遇到这段代码时，它会检查宏DEBUG是否已定义。如果是，它就会将fprintf()那行交给编译器，否则，就跳过该行，继续处理#endif后的代码。

```
#ifdef DEBUG
  fprintf(stderr, "This is a debugging message.\n");
#endif
```

注解　C预处理器并不懂C的任何语法、变量、函数或其他元素，它只看宏和指令。

Unix上的C预处理器是cpp，你也可以用gcc -E来运行它。不过你一般很少需要单独运行预处理器。

15.1.3　连接库

C编译器对系统了解得不多，不足以创建出有用的程序。你需要额外加上一些库来创建完整的程序。所谓C库，就是一些已编译好的、通用的、可让你添加到自己程序的函数。例如，许多可执行程序都会用到数学库，因为其中包含了三角函数之类的东西。

库主要是在连接的时候（即连接器从对象文件产生可执行程序之时）发挥作用。比如说，你有一个需要用到gobject库的程序，但你忘了告诉编译器你需要连接它，那么你就会看到这样的错误信息：

```
badobject.o (.text+0x28): undefined reference to 'g_object_new'
```

这条报错信息的关键是g_object_new部分。当连接器检查badobject.o这个对象文件时，发现找不到g_object_new这个函数，于是就无法创建程序了。对于这个例子，你可以怀疑是你自己忘记加上gobject库了，因为错误信息说找不到的函数是g_object_new()。

> **注解**　undefined reference并不总是代表找不到库。还有可能是你漏了连接某个对象文件。不过区分库函数和你自己的函数应该是容易的。

要解决这个问题，首先你要知道gobject库在哪里，然后再用编译器的-l选项连接它。跟include文件相似，库分布在系统的各个地方（默认是在/usr/lib中），大多数会放在名为lib的子目录中。而对于上例，基本的gobject库文件是libgobject.a，而库的名字是gobject。所以完整的连接和编译应是这样：

```
$ cc -o badobject badobject.o -lgobject
```

如果库的所在地不在常规位置，你必须用-L选项来告诉连接器。假设badobject程序需要用到/usr/junk/lib中的libcrud.a，那么就应该这样编译：

```
$ cc -o badobject badobject.o -lgobject -L/usr/junk/lib -lcrud
```

> **注解**　如果你想在一个库中搜索特定的函数，可用nm，它会列出很多东西。比如试试这条：nm libgobject.a。（你可能要用locate命令来查找libgobject.a；现在很多发行版会将库放在/usr/lib特定的子目录中。）

15.1.4　共享库

名称以.a结尾的库（如libgobject.a）是静态库。当程序连接的是静态库时，连接器会将库文件中的机器码复制到你的程序中。于是，最终的可执行程序不需要该库也能运行起来，并且其所引用到的行为将不会改变。

然而，库是会一直变大的，就如同库会变多一样，所以复制静态库很浪费磁盘和内存空间。另外，如果某天你发现所用的静态库有问题，需要修改或替换，那些引用到它的程序就都要重新编译，才能使用到新的库。

共享库就没有这种问题。引用共享库的程序只会在需要时才将该库加载到内存中。而且多个进程可以共享内存中的同一个共享库。修改共享库的代码不需要重编译那些引用它的程序。

使用共享库的代价是管理困难、连接复杂。但是，只要明白以下四点，你就能搞定它：

- 如何列出程序需要的共享库；
- 程序如何查找共享库；
- 如何让程序连接共享库；
- 常见的共享库陷阱。

接下来的小节会告诉你怎样使用和维护你系统的共享库。如果你想大致了解共享库的工作原理或者了解连接器，你可以参考John R. Levine的著作*Linkers and Loaders*（Morgan Kaufmann，1999）和David M. Beazley、Brian D. Ward、Ian R. Cooke合著的*The Inside Story on Shared Libraries and Dynamic Loading*（Computing in Science & Engineering，2001），或者是Program Library HOWTO（http://dwheeler.com/program-library/）之类的在线资源。另外，ld.so(8)帮助手册也值得一读。

列出共享库的依赖关系

共享库与静态库通常放在同一个地方。Linux的两大标准库目录是/lib和/usr/lib。其中/lib是不应该包含静态库的。

共享库的名字的后缀中通常含有.so（意为共享对象），比如libc-2.15.so和libc.so.6。想看一个程序用到了什么共享库，可运行`ldd prog`，其中prog是程序名。以下用bash来举例：

```
$ ldd /bin/bash
    linux-gate.so.1 => (0xb7799000)
    libtinfo.so.5 => /lib/i386-linux-gnu/libtinfo.so.5 (0xb7765000)
    libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb7760000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75b5000)
    /lib/ld-linux.so.2 (0xb779a000)
```

考虑到最佳性能和灵活性，可执行程序本身通常是不知道它所用的共享库的所在位置的。它只知道共享库的名字，或只知道一点寻找共享库的提示。`ld.so`这个小程序（它是运行时动态连接器/加载器）可以为程序在运行时找到并加载共享库。以上`ldd`的输出中，左边是可执行程序所知道的库的名字，右边是`ld.so`所找到的库的位置。

输出的最后一行显示了`ld.so`的实际位置：`/lib/ld-linux.so.2`。

`ld.so`怎样找到共享库

共享库的一个常见问题是动态连接器无法找到库。如果可执行程序有预先配置好的运行时库搜索路径（runtime library search path，以下简称rpath）的话，则动态连接器一般会首先查找那里。很快你就会看到怎样创建这个路径。

接着，动态连接器就会参考系统缓存/etc/ld.so.cache，看看该库是否在常规的位置。这是从缓存配置文件/etc/ld.so.conf中的目录列表获取的库文件名字的快速缓存。

15

> **注解**　正如你所见的其他Linux配置文件一样，ld.so.conf可能会包含/etc/ld.so.conf.d中的配置文件。

在ld.so.conf里，每一行就是一个你要包含到缓存里的目录。这个列表通常很短，内容类似这样：

```
/lib/i686-linux-gnu
/usr/lib/i686-linux-gnu
```

标准的库目录/lib和/usr/lib是隐式的，即你不需要在/etc/ld.so.conf中包含它们。

如果你改动了ld.so.conf或者改变了某个共享库的目录，你都必须通过以下命令来手动重建/etc/ld.so.cache文件：

```
# ldconfig -v
```

-v选项会输出被ldconfig添加到缓存的目录的详细信息和它所监测到的改动。

ld.so找共享库时还会参考一个地方：环境变量LD_LIBRARY_PATH。我们很快就会讲到。

不要养成往/etc/ld.so.conf里乱塞东西的习惯。你应该清楚系统缓存中有哪些共享库，而如果你把杂七杂八的东西都放进去，你就会面临一个难以管理的系统，并可能有混淆的风险。如果你想给程序安排一个隐含的库路径，你可以使用内置的rpath。下面来看看怎么做。

把程序与共享库连接起来

假设你在/opt/obscure/lib中有一个叫libweird.so.1的动态库，并需要把它与myprog连接，可以这么做：

```
$ cc -o myprog myprog.o -Wl,-rpath=/opt/obscure/lib -L/opt/obscure/lib -lweird
```

-Wl、-rpath用于告诉连接器将某个目录包含到程序的rpath中。虽然写了-Wl、-rpath，但还是需要加上-L。

对于已编译的程序，可以使用patchelf来加入不同的rpath，不过最好还是在编译时就做好。

共享库的一些问题

共享库的灵活性和一些惊人的技巧会有被滥用的风险，进而可能让你的系统变得乱七八糟。可能导致的后果有以下三种：

- 找不到库；
- 性能低；
- 找错库。

出现共享库问题的头号原因来自环境变量LD_LIBRARY_PATH。把一些以冒号分隔的目录赋值到这个变量的话，就可令ld.so首先查找这些目录。如果你没有源码，或不能用patchelf，又或者你只是不想重新编译，你都可以用这招快速解决库移动后的依赖问题。但这可能会造成混乱。

永远都不要在启动文件中或在编译软件时设置LD_LIBRARY_PATH。动态运行时连接器看到这个变量时，就会对其中设定的目录进行库的搜索。这不仅会造成性能低下，而且更重要的是，它可能会导致库混淆，因为运行时连接器会为每一个程序到这些目录中查找库。

如果有一些没有源码的无足轻重的程序（又或者是一些你不想重编译的应用，如Mozilla等）迫使你用LD_LIBRARY_PATH来解决库依赖问题，那就将它嵌套进脚本里。假设你有一个可执行程序/opt/crummy/bin/crummy.bin，它需要/opt/crummy/lib中的共享库，你可以写一个类似这样的crummy

嵌套脚本：

```
#!/bin/sh
LD_LIBRARY_PATH=/opt/crummy/lib
export LD_LIBRARY_PATH
exec /opt/crummy/bin/crummy.bin $@
```

不使用LD_LIBRARY_PATH能避免大部分共享库问题。但开发者可能还会遇到一个偶尔出现的大问题，就是库的应用程序接口（Application Programming Interface，以下简称API）改变了，使得装好的软件都用不了。最好的解决方法就是预防。具体做法是：安装库时也使用-Wl、-rpath，或者使用静态库。

15.2 make

如果一个程序需要用到不止一个源码文件，或者需要在编译时加上一些奇怪的选项的话，那么手动编译就很麻烦了。这个问题曾经困扰了人们很久，直至Unix上出现了一个叫make的编译管理工具。Unix的使用者应该对make有所了解，因为有些系统工具是会用到它的。不过，本章只会谈到它的冰山一角。关于make的东西是可以写出一本书的，例如Robert Mecklenburg的*Managing Projects with GNU Make*（O'Reilly，2004）。此外，大多数的Linux包都是由封装过的make或类似的工具构建的。构建系统有很多，其中有个叫autotools的，我们会在第16章谈及。

make是一个很大的系统，但它并不难理解。当看到有叫Makefile或makefile的文件时，就说明你遇上make了。（试着运行make看你能构建什么东西。）

make背后的理念就是目标，即你想达到的目的。目标可以是文件（一个.o文件或一个可执行文件等等）或者标签。另外，有些目标是依赖于其他目标的。举例来说，在连接之前，你得先做好一堆.o文件。这种需求就是依赖关系。

make会根据一些规则（例如怎样把.c文件变成.o文件）来构建目标。make本身就有一些规则，但你也可以修改它，或增加自己的规则。

15.2.1 一个Makefile实例

看一下这个简单的Makefile，它会将aux.c和main.c构建成一个叫myprog的程序：

```
# object files
OBJS=aux.o main.o

all: myprog

myprog: $(OBJS)
	$(CC) -o myprog $(OBJS)
```

第一行开头的#表示该行是注释。

下面一行是宏定义，它把OBJS赋值为两个对象文件，这对于后续的操作很重要。现在，你要

知道如何定义宏，以及如何在以后再次使用它（$(OBJS)）。

接下来是第一个目标all。第一个目标就是运行make后所要达到的最终结果。

构建目标的规则写在冒号后面。对于all来说，就是满足一个叫myprog的东西。这就是本文件的第一个依赖关系：all依赖于myprog。注意，myprog可以是一个实际的文件，也可以是另一个规则的目标。在本例中，它两者都是（既是all的规则，也是OBJS的目标）。

为了构建myprog，这个Makefile在依赖关系中使用了$(OBJS)。该宏展开成aux.o和main.o，于是myprog就依赖于这两个文件了（它们必须是文件，因为该Makefile中没有其他名为aux.o或main.o的目标）。

该Makefile假设你有两个叫aux.c和main.c的C源码文件与其在同一目录中。执行make的话，就会产生以下输出，其中显示了make所运行的命令：

```
$ make
cc    -c -o aux.o aux.c
cc    -c -o main.o main.c
cc -o myprog aux.o main.o
```

图15-1展示了这些依赖关系。

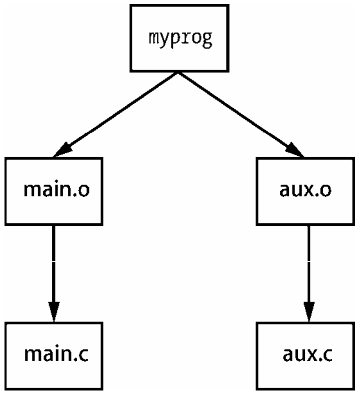


图15-1　Makefile依赖关系

15.2.2　内置规则

那么make是怎么知道要将aux.c变成aux.o的呢？不管怎么看，aux.c都没在Makefile中出现过。答案就是，make有自己内置的规则。当你需要.o文件时，它就会自动去找.c文件，它甚至懂得对那些.c文件运行cc -c命令，以达到获得.o文件的目标。

15.2.3　最终的程序构建

创建myprog的最后一步有点复杂，但其思想是很简单的。$(OBJS)有了两个对象文件之后，

你就可以按照最后一行来运行C编译器了（这里$(CC)展开成编译器的名字）：

```
    $(CC) -o myprog $(OBJS)
```

$(CC)前的空格是tab键。任何真实的命令前面都必须要有tab键。小心以下这种情况：

```
Makefile:7: *** missing separator. Stop.
```

这种错误是说Makefile损坏了。tab键就是分隔符。如果没有分隔符，或者出现其他干扰，你就会看到这样的错误提示。

15.2.4 保持更新

make的基础知识还有最后一点，就是目标需要跟它的依赖关系一同更新。如果你对上例make两次，那么第一次会构建出myprog，而第二次则会给出这样的信息：

```
make: Nothing to be done for 'all'.
```

在第二次时，make会发现规则中的myprog已经存在，而且自从上次构建之后，所有依赖关系都未曾改变，于是它就不会再次构建myprog。要解决这个问题，可按以下步骤执行。

(1) 执行touch aux.c。

(2) 再次执行make。这次，make发现aux.c比目录中已有的aux.o更新，于是它就会再次编译出aux.o。

(3) myprog是依赖于aux.o的，而现在aux.o比已有的myprog更新，于是它就会再次构建出myprog。

这是一种典型的反应链。

15.2.5 命令行参数与选项

熟悉make的命令行参数和选项的话，你会获得很多便利。

最有用的选项之一就是在命令行上指定一个单独的目标。对于上面例子，如果你只想得到aux.o的话，可以执行make aux.o。

你也可以在命令行定义一个宏。例如，想使用clang编译器的话，试试：

```
$ make CC=clang
```


这样，make就会使用你定义的CC来取代原本的编译器cc。命令行宏对于测试预处理器定义和库是很有用的，尤其是有CFLAGS和LDFLAGS时，这两个我们很快就会讲到。

实际上，运行make不一定要有Makefile。如果内置的make规则能完成目标，你可以直接叫make去执行目标。例如，你有一个叫blah.c的简单程序，试下make blah，它会这样做：

```
$ make blah
cc    blah.o -o blah
```

这种make只适用于最简单的C程序。如果你的程序需要用到库，或者需要特别的include目录，那你还是应该写个Makefile。在你不了解编译器的运作原理，而又想编译一些例如Fortran、Lex、Yacc之类的东西时，确实可以直接make而不用Makefile。为什么不试着让make帮你搞定呢？就算它做不到，它也会友善地提示你如何操作。

make还有以下两个好用的选项。

- -n：显示一次构建所要用到的命令，但并不执行它们。
- -f file：告诉make使用Makefile和makefile以外的文件。

15.2.6　标准宏和变量

make有很多特别的宏和变量。宏和变量的区别很难说清，所以，我们会把make启动以后不再改动的东西叫作宏。

如前文提到的，你可以在Makefile的开头设置宏。以下是最常见的一些宏。

- CFLAGS：C编译器选项。make会将这个选项作为参数，在将.c文件变成.o的阶段传给编译器。
- LDFLAGS：类似CFLAGS，不过它是在将.o变成可执行程序的阶段传给连接器。
- LDLIBS：如果你用了LDFLAGS，但不想库名选项与查找路径混在一起，可以将库名选项写在这里。
- CC：C编译器。默认是cc。
- CPPFLAGS：C预处理器选项。make运行预处理器时，将其作为参数。
- CXXFLAGS：GNU使用这个宏作为C++编译器选项。

make变量会随着目标的构建而改变。因为我们永远都只是在使用make变量，而不会去手动设置它们，所以把它们都带上$就好了。

- $@：写在规则里时，表示当前目标。
- $*：当前目标的基名。例如，在构建blah.o时，它表示为blah。

最完整的make变量列表在make的帮助手册中。

注解　注意，GNU的make有很多其他变种所没有的扩展、内置规则和特性。如果你只在Linux上使用，那没什么问题，但如果是迁到Solaris或BSD机器的话，就不一定能产生相同的效果了。好在我们有GNU autotools这类工具来解决跨平台的问题。

15.2.7　常规的目标

大部分的Makefile都包含了一些用于辅助编译的常规目标，如下所示。

- clean：这个目标无处不在。make clean通常会把所有对象文件和可执行程序都清掉，以便你重新构建或者打包软件。以下就是一个例子：

```
clean :
        rm -f $(OBJS) myprog
```

- distclean：GNU autotools所生成的Makefile总会有这个目标。它能删除原包以外的所有东西，包括Makefile。在第16章你会学到更多。偶尔你会发现有些开发者不喜欢用这个目标来清除可执行程序，而更喜欢用realclean。
- install：将文件和编译好的程序放到Makefile认为适当的地方。这可能会有风险，所以最好还是先用make -n install看看会放在哪里。
- test或check：有些开发者会加上test或check目标来检验构建出的东西是否可用。
- depend：通过编译器的-M选项来检查源码，以建立依赖关系。这是一个不寻常的目标，因为它经常会改动Makefile自身。这已经不是一种通用的做法了，但如果你遇到要求你这么做的情况，那最好还是照着去做。
- all：通常是Makefile的第一个目标。经常有人写all而不是写要构建的程序名。

15.2.8 组织一个Makefile

虽然Makefile的风格多样，但有些规范是大多数程序员都遵守的。其中一条就是，在Makefile的第一部分（宏定义）定义好不同用途的库和include：

```
MYPACKAGE_INCLUDES=-I/usr/local/include/mypackage
MYPACKAGE_LIB=-L/usr/local/lib/mypackage -lmypackage
PNG_INCLUDES=-I/usr/local/include
PNG_LIB=-L/usr/local/lib -lpng
```

于是各种编译器和连接器的选项就由这些宏组合而成：

```
CFLAGS=$(CFLAGS) $(MYPACKAGE_INCLUDES) $(PNG_INCLUDES)
LDFLAGS=$(LDFLAGS) $(MYPACKAGE_LIB) $(PNG_LIB)
```

可复用的对象文件则要定义好。例如，假设你的包会创建出名为boring和trite的可执行程序，它们有各自的.c文件，并且都需要用到util.c，则可以这样定义：

```
UTIL_OBJS=util.o

BORING_OBJS=$(UTIL_OBJS) boring.o
TRITE_OBJS=$(UTIL_OBJS) trite.o

PROGS=boring trite
```

Makefile的剩余部分就会是这样：

```
all: $(PROGS)

boring: $(BORING_OBJS)
	$(CC) -o $@ $(BORING_OBJS) $(LDFLAGS)

trite: $(TRITE_OBJS)
	$(CC) -o $@ $(TRITE_OBJS) $(LDFLAGS)
```

你可以将两个生成可执行程序的目标放在同一条规则里，但这不是一种好的做法，因为这样会使规则难以拆分和复用，甚至会造成错误的依赖关系：如果boring和trite处于同一条规则中，那么它们就都会依赖于对方的.c文件，这样的话，就算你只改动了其中一个.c，也会使make重新构建boring和trite。

注解　如果某个对象文件有特别的规则，就将该规则置于构建可执行程序的规则之上。如果多个可执行程序用到相同的对象文件，就将该对象文件的规则置于可执行程序的规则之上。

15.3　调试器

Linux上标准的调试器是gdb。除此之外，界面友好的Eclipse IDE和Emacs系统也是Linux支持的。如果希望你的程序产生完整的调试信息，可带-g选项运行编译器，以在程序中写入符号表和其他调试信息。以下是对名为program的可执行程序运行gdb：

```
$ gdb program
```

接着你会得到一个(gdb)提示符，然后可以这样传递命令行参数（假设参数为options）：

```
(gdb) run options
```

如果程序是可行的，它就会正常启动、运行、退出。而如果有问题的话，gdb就会停止，打印出错误的代码，并回到(gdb)提示符。因为打印出的代码通常就是问题的所在，所以可能需要将其中的变量也打印出来。（print命令也适用于数组和C结构体。）

```
(gdb) print variable
```

想要gdb在某段代码上暂停，可以使用断点功能。以下例子中，file是源码文件，line_num是需要暂停的位置：

```
(gdb) break file:line_num
```

想让gdb继续往下执行，可这样做：

```
(gdb) continue
```

想清除断点，输入：

```
(gdb) clear file:line_num
```

本节对gdb的介绍是极简单的，你可以在线获取更详尽的手册，或参考Richard M. Stallman et al.的*Debugging with GDB*, 10th edition（GNU Press，2011）。而关于调试则可看看 Norman Matloff和Peter Jay Salzman的*The Art of Debugging*（No Starch Press，2008）。

注解 如果想挖掘内存问题和生成统计信息，试试Valgrind（http://valgrind.org/）。

15.4 Lex 和 Yacc

如果你要编译的程序需要读取配置文件或命令，那你可能要用到Lex和Yacc。这两个工具是用于制作编程语言的。

- Lex是一个词法分析器的生成器，它能将文本内容转换成一个个标记。其GNU/Linux版本叫作flex。你可使用编译器的`-ll`或`-lfl`连接器标记来连接Lex的库。
- Yacc是一个语法分析器的生成器，能根据语法来读取标记。GNU的分析器是bison。为使生成的语法分析器与Yacc兼容，你需要执行`bison -y`。你可使用编译器的`-ly`连接器标记来连接Yacc的库。

15.5 脚本语言

以前，Unix管理员一般都不太关心Bourne shell和awk以外的脚本语言。现在shell脚本（第11章讲到的）依然是Unix的重要组成部分，而awk脚本则逐渐没落。另外，很多强大继任者的出现也使得不少系统编程从C转换到了脚本语言（如whois程序）。现在我们来看一些基本的脚本编程。

首先你要知道的是，所有脚本语言的第一行是跟Bourne shell的shebang类似的。例如，Python脚本的第一行大概是这样的：

```
#!/usr/bin/python
```

或者这样：

```
#!/usr/bin/env python
```

在Unix中，所有以`#!`开头的可执行文本文件都是脚本。其后的路径是该脚本的解释器。当Unix尝试运行以`#!`开头的可执行文件时，它会启动`#!`后的解释器，并将文件剩余的内容作为该解

释器的标准输入。所以，下例也是一个脚本：

```
#!/usr/bin/tail -2
This program won't print this line,
but it will print this line...
and this line, too.
```

shell脚本的第一行经常会出现这样的错误：脚本语言解释器的路径无效。举个例子，假设上面的脚本叫作myscript，如果tail不在/usr/bin中，而在/bin中，那么运行`myscript`将会产生以下报错信息：

```
bash: ./myscript: /usr/bin/tail: bad interpreter: No such file or directory
```

还有，不要期望解释器能接受多个参数。上例的`-2`是可行的，但如果再加多一个参数，系统就会将`-2`和第二个参数（包括空格）合为一个参数。不同的系统可能效果不一样，但最好还是别这么做。

下面来看一些具体的脚本语言。

15.5.1　Python

脚本语言Python拥有一系列强大的功能，如文本处理、数据库访问、网络编程、多线程等，而且支持者众多。它还有强大的交互模式和一套有组织的对象模型。

Python的可执行程序是python，通常在/usr/bin中。然而，它不仅能用于脚本编程，还可以用作建站工具。David M. Beazley的*Python Essential Reference*, 4th edition（Addison-Wesley，2009）开头有一段简短的教程，是一本不错的Python参考书。

15.5.2　Perl

Perl是Unix上的较为老旧的第三方脚本语言之一。它是编程界的“瑞士军刀”。虽然近年它被Python超越，但它仍是文本处理、转换、文件操作的利器，而且你会发现很多工具都是由它构建成的。Randal L. Schwartz、brian d foy和Tom Phoenix的*Learning Perl*, 6th edition（O’Reilly，2011）可当作教程来看。想更详细地了解Perl，可参考Chromatic的*Modern Perl*（Onyx Neon Press，2014）。

15.5.3　其他脚本语言

你可能还会见到以下这些脚本语言。

- PHP：它是超文本处理语言，常用于动态网页编程。也有些人只拿它当脚本用。其官网是http://www.php.net/。
- Ruby：面向对象的爱好者和Web开发者尤其喜欢这语言（http://www.ruby-lang.org/）。
- JavaScript：此语言主要是在浏览器中操作网页内容。大部分老程序员都觉得它缺点太多而不把它当作脚本语言，不过对于Web编程来说它几乎是必不可少的。它有一种实现叫作

Node.js，可执行程序是node。

- Emacs Lisp：它是Lisp语言的一个变种，在Emacs文本编辑器中使用。
- Matlab和Octave：Matlab是一套商业的矩阵及数学编程语言和库。Octave是类似Matlab的免费软件。
- R：一种流行的免费统计分析语言。详见http://www.r-project.org/和Norman Matloff的*The Art of R Programming*（No Starch Press，2011）。
- Mathematica：也是一套商业的数学编程语言和库。
- m4：宏处理语言，常见于GNU autotools。
- Tcl：Tcl（工具命令语言）是一种简单的脚本语言，其扩展有图形界面Tk和自动化工具Expect。虽然Tcl不再被广泛使用，但不要小看它的能力。很多经验丰富的开发者喜欢用Tk，尤其是喜欢用它做嵌入式开发。有关Tk，详见http://www.tcl.tk/。

15.6 Java

Java跟C一样都是编译型语言，它有更简单的语法和强大的面向对象能力。它在Unix上也较为常用。例如，它多用于制作Web应用和一些特定的应用。Android应用就通常是用Java来开发的。尽管我们很少在Linux桌面看到它，但你还是应该懂得Java的运作，至少是了解它如何在一个独立应用上运作。

Java编译器分为两种：用于生成机器码供系统使用的本地编译器（如C编译器）以及字节码解释器（有时也叫虚拟机，但不是第17章讲的那种虚拟机）使用的字节码编译器。你在Linux上看到的Java程序都是字节码。

Java字节码文件以.class结尾。Java运行时环境（Java Runtime Environment，以下简称JRE）包含了运行Java字节码所需的程序。想运行一个字节码文件，可以这样做：

```
$ java file.class
```

以.jar结尾的字节码文件也是有的，它由一堆.class文件打包而成。运行.jar文件需要用这种语法：

```
$ java -jar file.jar
```

有时你可能需要将Java的安装路径设置到`JAVA_HOME`环境变量中，甚至可能还需要使`CLASSPATH`变量包含你程序需要的所有class的所在目录。`CLASSPATH`是一个以冒号分隔的目录集合，看起来跟可执行程序所参考的`PATH`变量差不多。

你需要有Java开发工具（Java Development Kit，以下简称JDK）才能将.java文件编译成字节码。有了JDK，你就可以运行其中的`javac`编译器来创建.class文件：

```
$ javac file.java
```

JDK还包含`jar`程序，它能创建和拆分.jar文件，用法类似`tar`。

15

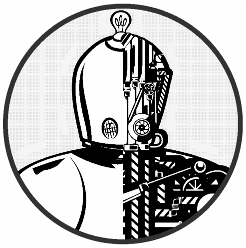
15.7　展望：编译包

编译器和脚本语言的世界很庞大，而且在不断地扩张。就在本书成书之际，新的编译型语言如Go（golang）和Swift也已逐渐流行起来。

LLVM编译器设备集（http://llvm.org/）能显著地简化编译器的开发。如果你对设计和实现编译器有兴趣，这两本书值得一看：Alfred V. Aho et al.的*Compilers: Principles, Techniques and Tools*, 2nd edition（Addison-Wesley，2006）和Dick Grune et al.的*Modern Compiler Design*, 2nd edition（Springer，2012）。而对于脚本语言，最好还是参考在线资源，因为脚本语言的实现五花八门。

掌握了编程工具的基础，你就可以去看看它们能做什么了。下一章讲的就是如何从源代码构建包。

第 16 章 从C代码编译出软件



大部分非专利的第三方Unix软件都是以源代码的形式放出，使人们能自行构建并安装。其中一个原因是，Unix（以及Linux）版本繁多，架构各异，我们很难造出符合各平台的二进制程序的组合包。另外，同样重要的是，Unix社区上放出的各式各样的源码鼓励用户为漏洞修复和功能添加做贡献，这使开源变得有意义。

Linux系统上，几乎所有的东西都有它的源代码——从内核、C库，到网页浏览器等等。你甚至可以使用源代码来（重新）安装你系统的某部分，来更新和加强你的系统。但是，你不应该让所有东西都从源代码构建并安装，除非你真的很享受这个过程或者某些特殊原因使然。

Linux发行版的核心部分，如/bin中的程序，一般都不难更新，而且，Linux的安全问题通常都修复得很快。然而，不要期望你的发行版能为你提供一切。以下这些原因就解释了为何你需要自行安装某些包。

- 你可按自己的需要进行设置。
- 你可自己决定安装位置。甚至你还可以安装同一个包的不同版本。
- 你可自行控制版本。发行版不一定会自带所有包的最新版本，尤其是那些附属的软件包（如Python的库）。
- 你可了解该包如何运作。

16.1　软件的构建系统

Linux上的编程环境有很多种：从传统的C语言到如Python的解释型脚本语言。它们各自都有至少一套区别于Linux发行版自带工具的构建和安装系统。

我们这一章会讲解如何通过其中的GNU autotools套件所产生的配置脚本来编译和安装C

代码。大家都普遍认为这套系统是稳定的，而且实际上很多基本的Linux工具都使用到了它。因为它是基于make等现有的工具之上的，所以看完本章之后，你可以将这些知识套在其他构建系统上。

从C代码安装一个软件，通常涉及这些步骤：

(1) 将源代码的归档解包；

(2) 对包进行设置；

(3) 运行make来构建程序；

(4) 运行make install或者发行版特定的安装命令来安装该包。

注解　在往下读之前，你应该先弄懂第15章讲的基础知识。

16.2　解开 C 源码包

一个软件包的源码包，通常是一个.tar.gz、.tar.bz2或.tar.xz文件，你需要按照2.18节的做法来解开它。不过在此之前，还是先用tar tvf或tar ztvf来检查下里面的内容，因为有些包解开时不会建立自己的目录。

如果输出是这样，那就可以解开它了：

```
package-1.23/Makefile.in
package-1.23/README
package-1.23/main.c
package-1.23/bar.c
--snip--
```

不过，它也有可能没有将文件放在一个目录（如上例的package-1.23）里：

```
Makefile
README
main.c
--snip--
```

将上面这种包直接解包的话，会使你的当前目录变得一团乱。为了避免这种情况，你要先创建一个目录，并cd进去，再在里面进行解包。最后一点，你还要留意那些包含绝对路径名的文件，例如这些：

```
/etc/passwd
/etc/inetd.conf
```

这种情况很少见，但如果你真的遇到了，请将该包删掉。里面可能包含木马或者一些恶意代码。

从哪里开始

当你解包完，并看到一堆文件出现时，请先去找找README和INSTALL文件。无论如何，README文件都是要首先看的，因为里面含有关于该包的描述、手册、安装提示，以及其他有用的信息。很多包都会提供INSTALL文件，里面是编译和安装软件的指令。请注意其中特别的编译器选项和定义。

除了README和INSTALL，你还会看到以下三类文件。

- 与`make`系统相关的文件，例如Makefile、Makefile.in、configure、CMakeList.txt等。有些旧的软件的Makefile可能需要你自己去改，但现在大多都会用GNU autoconf或CMake之类的配置工具。这些工具有其脚本或配置文件（如configure或CMakeList.txt），可根据你的系统设定和配置选项来帮你从Makefile.in中生成Makefile。
- 以.c、.h或.cc结尾的源码文件。包里到处都会有C源码文件。而C++源码文件则通常以.cc、.C或.cxx结尾。
- 以.o结尾的对象文件，或者二进制文件。一般来说，源码包中是没有对象文件的，但如果该包的维护者无权释放源代码而只能提供对象文件的话，那你就要自己处理它们了。大多数情况下，源码包里含有对象文件（或可执行的二进制文件），就意味着该软件打包有问题，你需要运行`make clean`来进行全新的编译。

16.3 GNU autoconf

虽然C代码一般都是可移植的，但我们还是难以只靠一个Makefile来适应各平台的差异。早期的解决方法是为不同的操作系统提供不同的Makefile，或者是提供一个易于修改的Makefile。这种做法后来演变成了使用脚本来分析系统（这种分析以往只用于构建软件包），再产生出Makefile。

GNU autoconf就是一套流行的、用于自动产生Makefile的系统。使用此套系统的包都会带有configure、Makefile.in和config.h.in文件。其中.in文件是模板。它的做法是，运行`configure`脚本来分析你系统的特性，然后在.in文件的基础上做一些替换，最后创建出真正的构建文件。对于终端用户来说，这个过程是简单的，只需要像下面这样执行`configure`就能从Makefile.in产生出Makefile：

```
$ ./configure
```

因为该脚本会先检查你的系统，所以它会输出一大堆诊断信息。如果一切正常，`configure`就会创建出一个或多个Makefile、一个config.h文件以及一个缓存文件config.cache。缓存文件能使你下次`configure`时不必再做一次系统检查。

现在你可以运行`make`来编译那个包了。虽然`configure`成功不代表`make`也能成功，但它作为第一步来说还是很重要的。（有关`configure`和编译失败的问题查找，见16.6节。）

16

下面来亲自体验下这个过程吧。

注解　现在你需要集齐一些必要的构建工具。对于Debian和Ubuntu来说，最简单的做法就是安装build-essential包；而对于类Fedora系统，请用第15章的开发工具groupinstall。

16.3.1　一个autoconf的例子

在讨论自定义autoconf的行为之前，我们先看一个普通的例子，这样你才知道你想要自定义的是什么。你需要先把GNU coreutils包安装在自己的root目录里（以确保不会影响整个系统）。该包可在http://ftp.gnu.org/gnu/coreutils/（通常最新的就是最好的）获取，然后解开它，进入它的目录中，进行以下配置：

```
$ ./configure --prefix=$HOME/mycoreutils
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
--snip--
config.status: executing po-directories commands
config.status: creating po/POTFILES
config.status: creating po/Makefile
```

接着make它：

```
$ make
  GEN      lib/alloca.h
  GEN      lib/c++defs.h
--snip--
make[2]: Leaving directory '/home/juser/coreutils-8.22/gnulib-tests'
make[1]: Leaving directory '/home/juser/coreutils-8.22'
```

下一步试着运行某个刚刚创建出的可执行文件，如./src/ls，再试着运行`make check`，来对该包进行一系列的检查。（这会花一些时间，但是很有趣。）

最后，可以安装该包了。先用-n选项空跑一次，看看它准备安装些什么：

```
$ make -n install
```

检查一下输出，如果没有什么异常（例如它准备安装在mycoreutils以外的地方），就实施安装：

```
$ make install
```

现在你root目录里应该有一个子目录叫mycoreutils，里面应该有了bin、share等子目录。看看bin里的那些程序（你刚刚建好了第2章里提到的很多基本工具）。最后一点，因为你将mycoreutils放在了你的root目录下，它与你系统的其他部分独立了开来，所以你可随意将其删掉，而不用担

心对系统造成损害。

16.3.2　使用打包工具来安装

大部分Linux发行版都带有软件打包工具，它创建出的安装包能对由该包安装的软件进行后期维护。基于Debian的发行版（如Ubuntu）或许是最简单的，就像下面这样，使用checkinstall，而不单是使用make install：

```
# checkinstall make install
```

你可以用--pkgname=name选项来指定新包的名称。

创建RPM包会更复杂一点，因为你得先创建一个目录树以便制作包。你可以使用rpmdev-setuptree命令来实现，然后再用rpmbuild工具来完成剩下的工作。最好还是按照网上的教程来做。

16.3.3　configure脚本的选项

刚才你已看到了configure脚本最有用的选项之一：使用--prefix来指定安装位置。autoconf默认生成的Makefile中的install目标都是使用/usr/local作为前缀：二进制程序会去到/usr/local/bin，库会去到/usr/local/lib，等等。更改前缀可执行如下命令：

```
$ ./configure --prefix=new_prefix
```

configure的大多数版本都有--help选项，可以列出其他配置选项。不幸的是，该列表实在太长了，难以得知哪些是重点。以下我们特地列举了一些必要的选项。

- --bindir=directory：将可执行程序装在directory目录。
- --sbindir=directory：将系统级的可执行程序装在directory目录。
- --libdir=directory：将库装在directory目录。
- --disable-shared：不构建共享库（要看具体是什么库）。不构建的话，或许能避免一些后续的麻烦。（详见15.1.4节。）
- --with-package=directory：告诉configure需要用到directory目录的包。当某个库不在标准位置时，这个选项是比较好用的。但不幸的是，并非所有的configure脚本都能识别这个选项，而且，它的语法不明确。

使用不同的构建目录

如果你想试试上述这些选项的话，你可以在用另一个目录构建时进行尝试。首先在任意一个地方新建一个目录，然后在新目录运行原目录的configure脚本。这样所产生的Makefile将会依然使用原目录的源码，但make出的东西却留在新目录。（有些开发者更希望你这么做，因为这样不仅不会在原目录产生新的东西，而且还有利于使用相同的源码为不同平台或使用不同配置选项进行构建。）

16.3.4　环境变量

你可以通过修改一些会被configure脚本当成make变量的环境变量来影响configure的行为。最重要的环境变量是CPPFLAGS、CFLAGS和LDFLAGS。但要小心的是，configure对环境变量是很挑剔的。比如说，对于头文件目录，你应该使用CPPFLAGS而不是CFLAGS，因为configure经常会单独运行预处理器。

在bash中，为configure设置环境变量的最简单的做法就是在./configure前放置变量的声明。例如，以下命令就为预处理器定义了宏DEBUG：

```
$ CPPFLAGS=-DDEBUG ./configure
```

注解　你也可以用选项的形式来传递变量，例如：

```
$ ./configure CPPFLAGS=-DDEBUG
```

使用环境变量来指引configure查找第三方include文件和库也是很方便的。例如，以下命令会使预处理器到include_dir里查找：

```
$ CPPFLAGS=-Iinclude_dir ./configure
```

如15.2.6节所讲，要使连接器到lib_dir里查找，可用此命令：

```
$ LDFLAGS=-Llib_dir ./configure
```

如果要用到lib_dir的共享库（见15.1.4节），那么上例是设置不了运行时动态连接的。你还需要用到连接器的-rpath选项：

```
$ LDFLAGS="-Llib_dir -Wl,-rpath=lib_dir" ./configure
```

设置变量时要谨慎。一点差错就会使configure失败。比如，假设你的-I写少了一杠，像下面这样：

```
$ CPPFLAGS=Iinclude_dir ./configure
```

就会产生这样的报错：

```
configure: error: C compiler cannot create executables
See 'config.log' for more details
```

查看这次失败所产出的config.log，你会发现下列信息：

```
configure:5037: checking whether the C compiler works
configure:5059: gcc Iinclude_dir   conftest.c >&5
gcc: error: Iinclude_dir: No such file or directory
configure:5063: $? = 1
configure:5101: result: no
```

16.3.5 autoconf的目标

若configure成功执行，你会发现它所生成的Makefile里除了有标准的all和install，还包含如下所列的一些有用的目标。

- make clean：如第15章所述，它会清除所有对象文件、可执行程序和库。
- make distclean：它与make clean很像，只不过它清除的是所有自动产生的东西，包括Makefile、config.h、config.log等等。也就是说，它使整个源码目录就像刚解包出来一样。
- make check：有些包会自带一些用于检查所编译出的程序是否正确的测验；而make check就运行这些测验。
- make install-strip：它与make install很像，只不过是在安装时，它会将可执行程序和库内的符号表及其他调试信息都移除掉。移除之后，程序占的空间会少很多。

16.3.6 autoconf的日志文件

如果configure的执行过程出了错，但你却看不出哪里有错，那么你可以检查一下config.log。不幸的是，config.log通常是很大的，不容易定位问题的根源。

一般的做法是去到config.log的底部（比如在less里按G键），然后不断往上翻页，直到看见问题。然而，这也是很麻烦的，因为configure会将所有环境信息包括输出变量、缓存变量和其他定义都写在那里。所以，与其从底部往上翻页，不如从底部往上搜索“for more details”之类的字符串，或configure报错的内容。（记住，你可在less中使用?命令来发起反向搜索。）很有可能错误就在你搜到的内容中。

16.3.7 pkg-config

第三方的库有很多，如果都放在同一个地方，那会显得很乱。但是，如果各自放在单独的地方，那么在连接时又会出现麻烦。例如，假设你要编译OpenSSH，它需要用到OpenSSL库，那么你该怎样将OpenSSL库的位置告知OpenSSH的configure呢？

现在很多库都用pkg-config来解决问题，它不仅可以公告include文件和库的位置，还可以用于明确指定编译和连接的选项。其语法如下：

```
$ pkg-config options package1 package2 ...
```

例如，查找OpneSSL所需的库，可用以下命令：

```
$ pkg-config --libs openssl
```

其输出大概是这样：

```
-lssl -lcrypto
```

想查看pkg_config所知的全部库，用以下命令：

```
$ pkg-config --list-all
```

pkg-config的运作方式

探究其内幕，你会发现pkg_config是通过读取.pc配置文件来获取包信息的。例如，以下是Ubuntu中OpenSSL套接字库的openssl.pc（在/usr/lib/i386-linux-gnu/pkgconfig中）：

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib/i386-linux-gnu
includedir=${prefix}/include

Name: OpenSSL
Description: Secure Sockets Layer and cryptography libraries and tools
Version: 1.0.1
Requires:
Libs: -L${libdir} -lssl -lcrypto
Libs.private: -ldl -lz
Cflags: -I${includedir} exec_prefix=${prefix}
```

你可以修改这个文件，例如，给库选项增加-Wl,-rpath=${libdir}，以指定运行时动态连接的路径。但一个问题是，pkg-config是怎么找到.pc文件的呢？默认地，pkg-config会在其安装位置前缀的lib/pkgconfig目录里找。例如，如果安装位置前缀是/usr/local，那么它就会去/usr/local/lib/pkgconfig里找。

使用标准位置以外的pkg-config文件

不幸的是，pkg-config不会在标准位置以外的地方查找.pc文件。如果一个.pc文件的所在位置不标准，如/opt/openssl/lib/pkgconfig/openssl.pc，那么常规安装的pkg-config是不会读取到它的。以下是两个基本的解决方法。

- 将.pc文件的符号链接（或副本）集中到pkgconfig目录中。
- 使环境变量PKG_CONFIG_PATH包含那些另外的pkgconfig目录。环境变量只在本shell及子shell内有效。

16.4　实践安装

知道如何构建和安装软件固然很好，但更重要的是知道在何时与何处安装自己的软件。Linux发行版本身就带有很多软件,不过你最好看下是否你自己安装会更好。以下是自行安装的一些好处。

- 你可以进行一些自定义设置。
- 手动安装的话，会更好地理解该软件的用法。

- 想装什么版，就装什么版。
- 能更方便地对定制过的软件包进行备份。
- 能更方便地通过网络分享定制过的软件包（只要其架构是一致的且安装位置是相对独立的）。

以下是坏处。

- 这样比较花时间。
- 自行安装的软件包不会自动更新。发行版能毫不费力地管理软件的自动更新。网络应用的安全更新尤其重要。
- 如果你不用该软件，那么安装它只是浪费时间。
- 有可能装错。

就如早前构建coreutils（`ls`、`cat`等）时所见，安装软件没有多少难点，除非你定制得太多。不过，如果你对网络服务器如Apache很有兴趣，想完全掌控，那最好就是自行安装了。

在哪里安装

GNU autoconf及很多其他软件包的默认前缀都是/usr/local，它是本地安装软件的传统位置。操作系统不会更新/usr/local里的软件，所以自动更新不会使你那里的东西丢失。不过如果你自行安装的软件太多，那就会导致混乱。里面成千上万的零散文件，会使你无法理清它们各自属于哪里。

如果情况真的变得很糟糕，那么你就应该按16.3.2节所述来创建自己的软件包了。

16.5 打补丁

现在大多数的软件源码修改都可以通过在线版本库（如git库）的分支功能实现。但我们偶尔还是会遇到打补丁的做法，即使用补丁文件来修复漏洞和增加功能。你可能还听说过有人将diff等同于补丁文件，因为补丁文件是由diff程序产生的。

补丁文件的开头类似这样：

```
--- src/file.c.orig     2015-07-17 14:29:12.000000000 +0100
+++ src/file.c   2015-09-18 10:22:17.000000000 +0100
@@ -2,16 +2,12 @@
```

补丁里所记录的改动可以来自多个文件。你可以在补丁里搜索“三杠”（---）来得知修改了哪些文件。你还应注意补丁开头所示的工作目录。如上例所指的是src/file.c，所以你应该在执行补丁之前先去到含有src目录的目录，而非去到src目录。

打补丁要用到patch命令：

```
$ patch -p0 < patch_file
```

16

如果一切顺利，patch就会将文件都更新好，并正常退出。不过，如果出现这样的提示：

```
File to patch:
```

那就很可能是你进错了目录，或者是该目录里的源码与补丁文件里的不匹配。这就有点棘手了：它可能会导致有些代码更新了，有些没更新，使得接下来的编译不成功。

有时你可能会遇到这种情况：

```
--- package-3.42/src/file.c.orig      2015-07-17 14:29:12.000000000 +0100
+++ package-3.42/src/file.c    2015-09-18 10:22:17.000000000 +0100
```

如果补丁开头与你当前环境有一点点不同（例如你改了包名），你可以让patch忽略路径中的第一个目录。比如说，你现在处于一个含有src目录的目录之中，但该目录不叫package-3.42，那你可以用-p1来忽略这个开头的目录名：

```
$ patch -p1 < patch_file
```

16.6　编译和安装的问题排查

如果你懂得区分编译器错误、编译器警告、连接器错误和共享库问题，那你应该能应付构建软件时出现的很多问题。本节会介绍一些常见的问题。虽然用autoconf的话是不太可能遇到这些问题的，但了解一下也无妨。

在进入细节之前，先确认自己能读懂几种make输出。学会区分错误与被忽略的错误是很重要的。以下就是一个需要你检查的真正的错误：

```
make: *** [target] Error 1
```

然而，有些错误提示则是Makefile怀疑出错了，但即使出错也无害。像这类提示你可以忽略：

```
make: *** [target] Error 1 (ignored)
```

还有，大型的包中经常会出现GNUmake多次调用自身的情况。这时，每次make都会带有一个[N]，其中N是个数字。通常你能够很快地在编译出错的提示中找到错误所在，例如：

```
[compiler error message involving file.c]
make[3]: *** [file.o] Error 1
make[3]: Leaving directory '/home/src/package-5.0/src'
make[2]: *** [all] Error 2
make[2]: Leaving directory '/home/src/package-5.0/src'
make[1]: *** [all-recursive] Error 1
make[1]: Leaving directory '/home/src/package-5.0/'
make: *** [all] Error 2
```

前三行就能告诉你问题出在/home/src/package-5.0/src的file.c。但麻烦的是，还有很多其他输出信息，使我们难以发现重点。所以，学会过滤以下所列举的make错误，能极大地帮助我们发掘真正的问题。

具体错误

以下列举了一些你可能会遇到的常见构建错误及其解释与修复途径。

问题

编译器错误信息：

```
src.c:22: conflicting types for 'item'
/usr/include/file.h:47: previous declaration of 'item'
```

解释与修复

src.c的第22行有对item的重复声明。你将该行移除（使用注释、#ifdef等等），即可修复。

问题

编译器错误信息：

```
src.c:37: 'time_t' undeclared (first use this function)
--snip--
src.c:37: parse error before '...'
```

解释与修复

缺少必要的头文件。最好是从帮助手册去获知需要什么头文件。首先，看看出错的那行（本例中是src.c的第37行）。它可能会是一个变量的声明，类似这样的：

```
time_t v1;
```

向下搜索v1，看它是怎么与函数一起使用的，例如：

```
v1 = time(NULL);
```

现在执行`man 2 time`或`man 3 time`，查找名为`time()`的系统调用和库调用。于是，在手册2中你找到了你所需要的内容，如下所示：

```
SYNOPSIS
     #include <time.h>

     time_t time(time_t *t);
```

这意味着`time()`需要time.h。所以，请把`#include <time.h>`置于src.c的开头，再编译一次。

16

问题

编译器（预处理器）错误信息：

```
src.c:4: pkg.h: No such file or directory
(long list of errors follows)
```

解释与修复

编译器对src.c运行C预处理器时，找不到include文件pkg.h。可能是有个库没安装好，或者include文件在非常规位置，需要你指明。通常，你只需要为预处理器选项（CPPFLAGS）加上-I这一include路径选项。（你可能还需要一个连接器选项-L。）

如果出错原因不是找不到库，那还有一种可能，即该操作系统不支持这个代码。你可以查看一下Makefile和README中关于平台的要求。

如果你的发行版是基于Debian的，试试用apt-file命令来查找头文件：

```
$ apt-file search pkg.h
```

它可能会帮你找到该包的开发版。而对于有yum的系统，则可以这样做：

```
$ yum provides */pkg.h
```

问题

make错误信息：

```
make: prog: Command not found
```

解释与修复

你要有prog程序才能构建该软件。如果prog是cc、gcc或ld之类，那就说明你系统上没有安装开发工具。而如果你觉得你已安装了，那就试试在Makefile中指明prog的完整路径。

还有一种不常见的情况是make即时构建并使用prog。如果你的$PATH包含当前目录（.）的话，那这是可以做到的。而如果你的$PATH不包含当前目录，你可以将Makefile的prog改成./prog，或者暂时将.加到$PATH中。

16.7　前瞻

至此我们讲解的还只是软件构建的基础。当你上手后，可继续探索以下话题。

- 学习autoconf以外的构建系统，如CMake和SCons。
- 为自己的软件打造一套构建系统。如果你在写软件，那你就需要挑选一套构建系统，并学会使用它。如果选择GNU autoconf的话，可参考John Calcote的著作*Autotools*（No Starch Press，2010)。
- 编译Linux内核。内核的构建系统与别的工具完全不同。它有一套方便你定制内核与模块的配置系统。不过，其过程是明了的，而且，如果你理解引导装载程序的运作，那应该不难做到。然而，编译内核还是要小心，要确保留有旧的内核，以防新的开不了机。

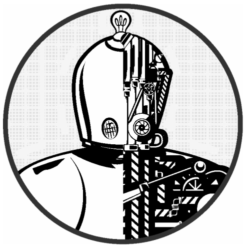
- 发行版专有的源码包。Linux发行版会各自持有软件源码作为其特殊的源码包。有时你可从它们那里获得功能扩展或漏洞修复的补丁。这些源码管理系统都包含了自动构建的工具，例如Debian有`debuild`，RPM有`mock`。

构建软件是学习编程和软件开发的基础。刚讲完的这两章内容揭示了你系统中软件的来源。下一步你应该可以很轻松地查看源码、修改源码，并制作出属于自己的软件。

第17章

在基础上搭建

17



之前的章节已经介绍过了Linux系统的各个基础组件，从底层的内核和进程组织到网络，再到一些构建软件的工具。学完这些以后，你能做什么呢？实际上，可以做很多东西。因为Linux几乎支持所有非专利的编程环境，所以可运行的应用程序自然种类繁多。下面来了解一下Linux擅长的一些应用领域，并看看你从本书学到的知识是如何联系起来的。

17.1 Web服务器与应用

Linux常用于运行Web服务器，而Linux上的应用服务器之王是Apache HTTP Server（简称Apache）。另外还有一个你会经常听说的，就是Tomcat（也来自Apache项目），它能支持基于Java的应用。

Web服务器本身做的事情不多——就是提供文件服务。Apache等大多数Web服务器的目标都是为Web应用提供底层平台。例如，Wikipedia是来自MediaWiki包的，它能让你建立自己的wiki。而Wordpress和Drupal之类的内容管理系统则可以让你建立自己的博客和媒体网站。构建这些应用的编程语言都能在Linux上使用。例如，MediaWiki、Wordpress和Drupal都是用PHP编写的。

这些Web应用的每一部分都是高度模块化的。你可以方便地添加自己的扩展，并且还可以通过Django、Flask和Rails之类的框架来创建自己的应用，它们能助你轻松地做出通用的Web基础设施和功能，例如模板、多用户、数据库支持等。

要让Web服务器正常发挥，你必须先有坚实的操作系统作为基础。在这方面，第8章到第10章的知识尤其重要。你的网络配置必须是无懈可击的，但更重要的是，你要懂得资源管理。大小适当、性能优越的内存和磁盘是必要元素，特别是在你的应用需要搭配数据库的时候。

17.2 数据库

数据库是存储和查阅数据的专业工具，而Linux上可用的数据库有很多种。数据库受人喜爱的地方有两点：它管理数据的方式简单统一，而且操作高效。

有了数据库，查询和修改数据变得更方便，尤其是相对于文本解析和修改来说。例如，在网络上使用/etc/passwd和/etc/shadow是很麻烦的。解决方法是，你可以建立一个提供用户信息的LDAP（Lightweight Directory Access Protocol，轻量目录访问协议）数据库来支持Linux的认证系统。客户端的配置是很简单的，你只需编辑一下/etc/nsswitch.conf文件并加一些额外的配置即可。

数据库之所以能高效地读取数据，是因为它采用索引来记录数据的位置。比如你有一堆数据，里面是姓、名、电话号码的对应关系。你可以在任意属性（比如姓）上建立索引，然后，当你要按姓来查找一个人的时候，数据库一般就会先从索引那里筛选出正确的位置，而非直接遍历整个数据集。

数据库的种类

数据库有两种基本形式：关系型和非关系型。*关系型数据库*（也叫关系型数据库管理系统，或简称RDBMS）如MySQL、PostgreSQL、Oracle和MariaDB都是能将不同的数据连接在一起的多功能数据库。举个例子，你有两组数据，一组是邮编和人名，另一组是邮编和对应的州名。关系型数据库能快速帮你查出某个州的所有人名。我们一般用SQL（结构化查询语言）来跟数据库沟通。

非关系型数据库，有时也叫NoSQL数据库，是用于解决关系型数据库难以处理的问题。例如，文档存储型数据库（如MongoDB）用于简化整个文档的存储和索引的建立。键值数据库（如redis）则关注存取性能问题。NoSQL数据库没有像SQL这样的通用查询语言，而是需要通过各种各样的接口和命令来实现沟通。

第8章谈到的硬盘和内存性能问题对于大部分的数据库实现来说都是非常重要的，因为你需要权衡RAM（RAM的存取更快速）和硬盘各要存储多少数据。很多大型数据库系统还涉及网络问题，因为它可能分布在多个服务器上。最常见的网络装置就是*复制*（replication），其做法就是将一个数据库复制到多个服务器上，使该系统能够接受更多的客户端连接。

17.3 虚拟化

在大部分的组织中，给某套硬件分配特定的任务是不划算的。把针对某件任务而定制的操作系统安装到一台服务器上，会使得该服务器难以支持其他任务，除非你重新安装其他系统。而虚拟机技术则可以支持在同一套硬件上安装多于一个操作系统（被称为*来宾*），并能让你随心所欲地启动和关闭它们。你甚至还可以把一个虚拟机从一台机器复制到另一台机器上。

Linux可用的虚拟机有很多，如内核的KVM（内核虚拟机）和Xen。虚拟机能方便Web服务器和数据库服务器的管理。虽然用单个Apache服务器支撑多个网站是可以的，但这样并不灵活，

而且不便维护。如果那些网站有不同用户运行的话，你就要同时维护这些服务器和用户。相反，我们更主张在单个物理服务器上建立多个虚拟机，并分配给各个用户，这样他们就不会相互干扰，而你也可以随意修改或移动它们。

管理虚拟机的软件叫作hypervisor。hypervisor能操控本书提及的Linux底层的各个部分，令虚拟机的使用与物理机无异。

17.4　分布式计算与实时计算

为了简化本地资源的管理，你可以在虚拟机技术之上搭建一些成熟的工具。*云计算*就是关于这方面的一个笼统的术语。其中的*基础设施即服务*（以下简称IaaS）就是一种能让你指定和使用远端CPU、内存、存储和网络等基础计算资源的系统。OpenStack项目就是这样一个包含了IaaS的接口平台。

更进一步，你还可以指定设施上的操作系统、数据库服务器、Web服务器之类的平台资源。这种级别的服务叫作*平台即服务*（以下简称PaaS）。

Linux是这些服务的核心，因为它经常被用作这些服务的底层操作系统。你在本书看到的几乎所有东西，包括内核，都能体现在这些系统上。

17.5　嵌入式系统

诸如音乐播放器、视频播放器、恒温器等提供特定用途的系统都是嵌入式系统。你可拿它跟桌面或服务器系统（能做各种事情但并不精于某一项的系统）对比一下。

你可以认为嵌入式系统是与分布式系统相反的，它不但没有打算扩展系统的规模，反而总是希望缩减它，以放进一个小的设备中。当今最流行的嵌入式Linux应该是Android了。

嵌入式系统通常需要由特定的硬件跟软件组合而成。比如说，你可以为PC添加足够的网络硬件以及配置正确的网络环境，来使其成为一个无线路由器。但更常见的做法是买一个抛除多余硬件而只保留必要硬件的小型专用路由器设备。例如，路由器与桌面机器相比需要更多的网络端口，而完全不需要视频或音频硬件。有了定制的硬件之后，还要有定制的软件，例如定制操作系统和用户界面。第9章提到的OpenWRT就是一个这样的Linux定制发行版。

随着越来越多的小型硬件的面世，嵌入式系统也越发受到关注，尤其是能将处理器、内存和外围设备集成在一小块空间上的单片系统。如Raspberry Pi和BeagleBone之类的单板系统就是这种设计，它们可选择一些Linux变种来作为操作系统。这些设备提供容易接收的输出以及传感器输入，可用Python等语言来操作，使得它们能广泛用于原型设计和小工具的制作。

各个嵌入式Linux的差异在于从服务器/桌面版保留了多少功能。配置有限的小型设备必须将基本功能以外的东西拿掉，以节省空间。也就是说，甚至可能连shell和一些核心工具都要由BusyBox来提供。这些系统与完整安装的Linux有很大不同，而且你可能会在上面看到一些老旧的软件，如System V init。

供嵌入式系统使用的软件，其开发平台一般还是在桌面环境上。而一些更强大的设备如Raspberry Pi则拥有更丰富的存储空间和更强的能力来运行许多开发工具。

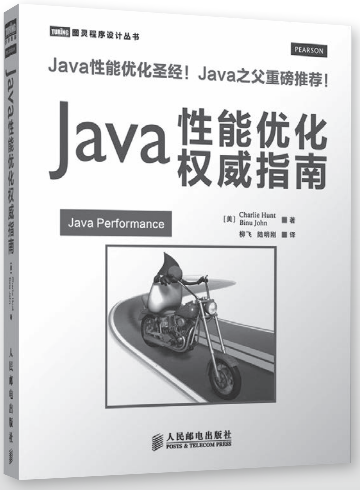
抛开不同的部分，嵌入式设备也具备本书描述的Linux基因：内核、设备、网络接口、init、用户进程。嵌入式的内核跟普通的内核相近（或者说一样），只是很多功能被屏蔽了。从底层越往上层走，才会越觉得它跟一般的Linux不同。

17.6 结束语

关于Linux，不论你的学习目标是什么，我都希望本书对你有益。而我的目标就是，让你有信心能在系统中修改或创建东西。至此，你应该能掌控你的系统了，现在就去亲身实践吧。

版权声明

延 展 阅 读

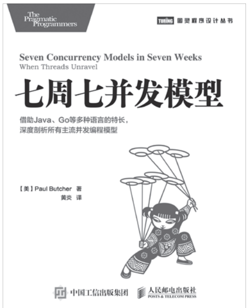


- **Java 性能优化圣经！**
- **Java 之父重磅推荐！**

《Java 性能优化权威指南》由曾任职于 Oracle/Sun 的性能优化专家编写，系统而详细地讲解了性能优化的各个方面，帮助你学习 Java 虚拟机的基本原理、掌握一些监控 Java 程序性能的工具，从而快速找到程序中的性能瓶颈，并有效改善程序的运行性能。

Java 性能优化的任何问题，都可以从本书中找到答案！

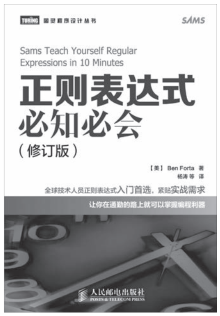
Java 性能优化权威指南
书号：978-7-115-34297-3
定价：109.00 元



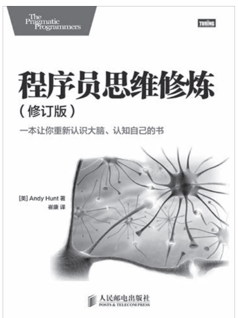
七周七并发模型
书号：978-7-115-38606-9
定价：49.00 元



发布！软件的设计与部署
书号：978-7-115-38045-6
定价：49.00 元



正则表达式必知必会（修订版）
书号：978-7-115-37799-9
定价：29.00 元



程序员思维修炼（修订版）
书号：978-7-115-37493-6
定价：49.00 元



高效程序员的 45 个习惯：敏捷开发修炼之道（修订版）
书号：978-7-115-37036-5
定价：45.00 元

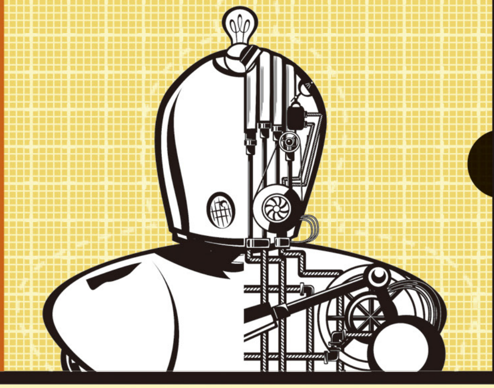


程序员健康指南
书号：978-7-115-36716-7
定价：39.00 元

How Linux Works

2nd Edition

What Every Superuser Should Know



Linux是了解操作系统工作机制的最佳平台。我们大多数人都用了多年电脑，但对电脑背后的工作机制却一无所知，而本书就是解除这一困惑的绝好途径。本书能让你对使用的计算机有所了解，熟悉软件的基本工作原理，以及理解系统作为一个整体是如何工作的。根据系统启动的大体顺序，本书深入介绍了从设备管理到网络配置的各个部分，演示了系统各部分的运行方式，并介绍了一些基本技巧和开发人员常用的工具。

我们学习Linux的原因可能各不相同。对于IT从业者（如系统运维人员）来说，他们需要了解本书中的几乎所有内容。对于Linux软件架构和开发人员来说，他们同样需要了解这些内容，以便发挥操作系统的最大功效。对于研究人员和学生来说，本书能够让他们理解为什么我们要这样设置系统。本书菜鸟与老鸟通吃，既可作为菜鸟的入门教程，也可以作为老鸟的进阶指南。

亚马逊读者评论

“本书绝不是枯燥的长篇大论，而是一本生动的Linux教程。Ward用严谨又不失风趣的语言和一个个鲜明易懂的例子，为你解除“谈命令行色变”的魔咒，让你非常直观地了解Linux系统背后的工作原理。”

“Brian Ward清晰明了地讲解了Linux内部的工作机制，从启动到磁盘、硬件等的各个方面。本书没有教你深奥的编程，却带你透彻领会系统各个层次是如何工作的，让你明白每一个动作执行的背后到底隐藏着何种玄机。”

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/操作系统

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-39492-7

ISBN 978-7-115-39492-7
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们:

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks